Assignment on **Series and Data frame**

Course name: Data Mining and Data Warehousing Lab

Course code: CSEL – 4108

Submitted By

Farhana Akter Suci

ID: B190305001

&

Rifah Sajida Deya

ID: B190305004

Submitted To

Dr. Md. Manowarul Islam

Associate Professor, Department of C S E, Jagannath University

August 29, 2024

Introduction

NumPy, Pandas, and Matplotlib are popular Python libraries used for scientific and analytical tasks. They make it easy to manipulate, transform, and visualize data efficiently. Pandas, which stands for **PANel DAta**, is a high-level tool for data analysis. It is easier to import and export data with Pandas. Built on top of NumPy and Matplotlib, Pandas provides a convenient platform for most data analysis and visualization tasks.

Data Structures in Pandas

A **data structure** is a system for organizing data, allowing for efficient storage, access, and modification of the data. It consists of data elements and the operations that can be performed on them.

There are 2 commonly used data structures in Pandas-

- Dataframe;
- Series

Further we are going to see creation and different operations of these data structures.

Dataframe

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. We can think of it like a spreadsheet or SQL table. It is generally the most commonly used pandas object.

Here, I'm working with more advanced features of DataFrame like sorting data, answering analytical questions using the data, cleaning data and applying different useful functions on the data.

Store the Result data in a DataFrame called StudentMarks.

```
import pandas as pd
StudentMarks= {
    'Name':['Suci','Suci','Suci','Srabanti','Srabanti','Srabanti', 'Ashravy','Ashravy','Ashravy','Mishti','Mishti','Mishti'],
 'UnitTest':[1,2,3,1,2,3,1,2,3,1,2,3],
 'DataMining':[22,21,14,20,23,22,23,24,12,15,18,17],
 'AI':[21,20,19,17,15,18,19,22,25,22,21,18],
 'Graphics':[18,17,15,22,21,19,20,24,19,25,25,20],
 'ImageProcessing':[20,22,24,24,25,23,15,17,21,22,24,25],
 'Cryptography':[21,24,23,19,15,13,22,21,23,22,23,20]
 }
```

```
MarksInformation=pd.DataFrame(StudentMarks)
MarksInformation
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	21	18	20	21
1	Suci	2	21	20	17	22	24
2	Suci	3	14	19	15	24	23
3	Srabanti	1	20	17	22	24	19
4	Srabanti	2	23	15	21	25	15
5	Srabanti	3	22	18	19	23	13
6	Ashravy	1	23	19	20	15	22
7	Ashravy	2	24	22	24	17	21
8	Ashravy	3	12	25	19	21	23
9	Mishti	1	15	22	25	22	22
10	Mishti	2	18	21	25	24	23
11	Mishti	3	17	18	20	25	20

Calculating Maximum Values

```
print(MarksInformation.max())
```

```
Name               Suci
UnitTest              3
DataMining           24
AI                   25
Graphics             25
ImageProcessing      25
Cryptography         24
dtype: object
```

If we want to output maximum value for the columns having only numeric values, then we can set the parameter numeric_only=True in the max() method, as shown below:

```
print(MarksInformation.max(numeric_only=True))
```

```
UnitTest            3
DataMining         24
AI                 25
Graphics           25
ImageProcessing    25
Cryptography       24
dtype: int64
```

Write the statements to output the maximum marks obtained in each subject in Unit Test 2.

```
UnitTest2 = MarksInformation[MarksInformation.UnitTest == 2]
print(f'\nResult of Unit Test 2:\n\n')
UnitTest2
```

```
Result of Unit Test 2:
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
1	Suci	2	21	20	17	22	24
4	Srabanti	2	23	15	21	25	15
7	Ashravy	2	24	22	24	17	21
10	Mishti	2	18	21	25	24	23

```
print(f'\nMaximum Mark obtained in Each Subject in Unit Test 2: \n\n{UnitTest2.max(numeric_only=True)}')
```

```
Maximum Mark obtained in Each Subject in Unit Test 2:
```

```
UnitTest           2
DataMining        24
AI                22
Graphics          25
ImageProcessing   25
Cryptography      24
dtype: int64
```

By default, the max() method finds the maximum value of each column (which means, axis=0). However, to find the maximum value of each row, we have to specify axis = 1 as its argument.

maximum marks for each student in each unit test among all the subjects

```
MarksInformation.max(numeric_only=True,axis=1)
```

```
0     22
1     24
2     24
3     24
4     25
5     23
6     23
7     24
8     25
9     25
10    25
11    25
dtype: int64
```

Calculating Minimum Values

```
print( MarksInformation.min(numeric_only=True))
```

```
UnitTest           1
DataMining        12
AI                15
Graphics          15
ImageProcessing   15
Cryptography      13
dtype: int64
```

```
MarksInformation.min()
```

```
Name               Ashravy
UnitTest                 1
DataMining              12
AI                      15
Graphics                15
ImageProcessing         15
Cryptography            13
dtype: object
```

Write the statements to display the minimum marks obtained by a particular student Suci in all the unit tests for each subject.

```
marksSuci = MarksInformation.loc[MarksInformation.Name == 'Suci']
print(f'\nMarks obtained by Suci in all the Unit Tests \n\n')
marksSuci
```

```
Marks obtained by Suci in all the Unit Tests
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	21	18	20	21
1	Suci	2	21	20	17	22	24
2	Suci	3	14	19	15	24	23

```
print(f'\nMinimum Marks obtained by Suci in each subject across the unittests\n\n{marksSuci[["DataMining", "AI", "Graphics", "ImageProcessir
```

```
Minimum Marks obtained by Suci in each subject across the unittests

DataMining        14
AI                19
Graphics          15
ImageProcessing   20
Cryptography      21
dtype: int64
```

Calculating Sum of Values

DataFrame.sum() will display the sum of the values from the DataFrame regardless of its datatype. The following line of code outputs the sum of each column of the DataFrame:

```
print(MarksInformation.sum())
```

```
Name               SuciSuciSuciSrabantiSrabantiSrabantiAshravyAsh...
UnitTest                                                          24
DataMining                                                       231
AI                                                               237
Graphics                                                         245
ImageProcessing                                                  262
Cryptography                                                     246
dtype: object
```

```
print(MarksInformation.sum(numeric_only=True))
```

```
UnitTest            24
DataMining         231
AI                 237
Graphics           245
ImageProcessing    262
Cryptography       246
dtype: int64
```

To print the sum of a particular column, we need to specify the column name in the call to function sum. The following statement prints the total marks of subject mathematics:

```
print(MarksInformation['DataMining'].sum())
```

```
231
```

Write the python statement to print the total marks secured by Suci in each subject.

```
marksRaman=MarksInformation[MarksInformation['Name']=='Suci']
print(f"\nMarks obtained by Suci in each test are:\n\n{marksRaman}")
```

```
Marks obtained by Suci in each test are:

   Name  UnitTest  DataMining  AI  Graphics  ImageProcessing  Cryptography
0  Suci         1          22  21        18               20            21
1  Suci         2          21  20        17               22            24
2  Suci         3          14  19        15               24            23
```

```
marksRaman[['DataMining','AI','Graphics','ImageProcessing','Cryptography']].sum()
```

```
DataMining        57
AI                60
Graphics          50
ImageProcessing   66
Cryptography      68
dtype: int64
```

To print total marks scored by Suci in all subjects in each Unit Test

```
marksSuci[['DataMining','AI','Graphics','ImageProcessing','Cryptography']].sum(axis=1)
```

```
0    102
1    104
2     95
dtype: int64
```

Calculating Number of Values

DataFrame.count() will display the total number of values for each column or row of a DataFrame. To count the rows we need to use the argument axis=1 as shown in the Program below.

```
print(MarksInformation.count())
```

```
Name               12
UnitTest           12
DataMining         12
AI                 12
Graphics           12
ImageProcessing    12
Cryptography       12
dtype: int64
```

Write a statement to count the number of values in a row.

```
MarksInformation.count(axis=1)
```

```
0     7
1     7
2     7
3     7
4     7
5     7
6     7
7     7
8     7
9     7
10    7
11    7
dtype: int64
```

Calculating Mean

DataFrame.mean() will display the mean (average) of the values of each column of a DataFrame. It is only applicable for numeric values.

```
MarksInformation.mean(numeric_only=True)
```

```
UnitTest            2.000000
DataMining         19.250000
AI                 19.750000
Graphics           20.416667
ImageProcessing    21.833333
Cryptography       20.500000
dtype: float64
```

Write the statements to get an average of marks obtained by Suci in all the Unit Tests.

```
MarksInformation
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	21	18	20	21
1	Suci	2	21	20	17	22	24
2	Suci	3	14	19	15	24	23
3	Srabanti	1	20	17	22	24	19
4	Srabanti	2	23	15	21	25	15
5	Srabanti	3	22	18	19	23	13
6	Ashravy	1	23	19	20	15	22
7	Ashravy	2	24	22	24	17	21
8	Ashravy	3	12	25	19	21	23
9	Mishti	1	15	22	25	22	22
10	Mishti	2	18	21	25	24	23
11	Mishti	3	17	18	20	25	20

```
Suci=MarksInformation[MarksInformation['Name']=='Suci']
SuciMarks =Suci.loc[:,'DataMining':'Cryptography']
print("\n\nSlicing of the DataFrame to get only the marks\n\n")
SuciMarks
```

```
Slicing of the DataFrame to get only the marks
```

	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	22	21	18	20	21
1	21	20	17	22	24
2	14	19	15	24	23

Average of marks obtained by Zuhaire in all Unit Tests

Average of marks obtained by Suci in all Unit Tests

```
SuciMarks.mean(axis=1)
```

```
0    20.4
1    20.8
2    19.0
dtype: float64
```

Calculating Median

DataFrame.Median() will display the middle value of the data. This function will display the median of the values of each column of a DataFrame. It is only applicable for numeric values.

```
print(MarksInformation.median(numeric_only=True))
```

```
UnitTest            2.0
DataMining         20.5
AI                 19.5
Graphics           20.0
ImageProcessing    22.5
Cryptography       21.5
dtype: float64
```

Write the statements to print the median marks of mathematics

```
DataMining=MarksInformation['DataMining']
```

```
DataMining
```

```
0     22
1     21
2     14
3     20
4     23
5     22
6     23
7     24
8     12
9     15
10    18
11    17
Name: DataMining, dtype: int64
```

```
DataMining1=DataMining[MarksInformation.UnitTest==1]
print("Displaying the marks scored in DataMining in UnitTest-1")
DataMining1
```

```
Displaying the marks scored in DataMining in UnitTest-1
0    22
3    20
6    23
9    15
Name: DataMining, dtype: int64
```

```
DataMiningMedian=DataMining1.median()
print("Displaying the median of Mathematics in UnitTest-1\n",DataMiningMedian)
```

```
Displaying the median of Mathematics in UnitTest-1
 21.0
```

Here, the number of values are even in number so two middle values are there i.e. 20 and 22. Hence, Median is the average of 20 and 22.

Calculating Mode

DateFrame.mode() will display the mode. The mode is defined as the value that appears the most number of times in a data. This function will display the mode of each column or row of the DataFrame. To get the mode of Hindi marks, the following statement can be used.

```
 MarksInformation['DataMining']
```

```
0     22
1     21
2     14
3     20
4     23
5     22
6     23
7     24
8     12
9     15
10    18
11    17
Name: DataMining, dtype: int64
```

```
MarksInformation['DataMining'].mode()
```

```
0    22
1    23
Name: DataMining, dtype: int64
```

Calculating Quartile

Dataframe.quantile() is used to get the quartiles. It will output the quartile of each column or row of the DataFrame in four parts i.e. the first quartile is 25% (parameter q = .25), the second quartile is 50% (Median), the third quartile is 75% (parameter q = .75). By default, it will display the second quantile (median) of all numeric values.

```
MarksInformation.quantile(numeric_only=True)
```

```
UnitTest            2.0
DataMining         20.5
AI                 19.5
Graphics           20.0
ImageProcessing    22.5
Cryptography       21.5
Name: 0.5, dtype: float64
```

By default, median is the output

```
MarksInformation.quantile(numeric_only=True,q=.25)
```

```
UnitTest            1.00
DataMining         16.50
AI                 18.00
Graphics           18.75
ImageProcessing    20.75
Cryptography       19.75
Name: 0.25, dtype: float64
```

```
MarksInformation.quantile(numeric_only=True,q=.75)
```

```
UnitTest            3.00
DataMining         22.25
AI                 21.25
Graphics           22.50
ImageProcessing    24.00
Cryptography       23.00
Name: 0.75, dtype: float64
```

Write the statement to display the first and third quartiles of all subjects.

```
Subjects=MarksInformation[['DataMining','AI','Graphics','ImageProcessing','Cryptography']]
print("Marks of all the subjects:\n\n")
Subjects
```

```
Marks of all the subjects:
```

	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	22	21	18	20	21
1	21	20	17	22	24
2	14	19	15	24	23
3	20	17	22	24	19
4	23	15	21	25	15
5	22	18	19	23	13
6	23	19	20	15	22
7	24	22	24	17	21
8	12	25	19	21	23
9	15	22	25	22	22
10	18	21	25	24	23
11	17	18	20	25	20

```
Quartiles=Subjects.quantile([.25,.75])
print("First and third quartiles of all the subjects:\n\n")
Quartiles
```

```
First and third quartiles of all the subjects:
```

	DataMining	AI	Graphics	ImageProcessing	Cryptography
0.25	16.50	18.00	18.75	20.75	19.75
0.75	22.25	21.25	22.50	24.00	23.00

Calculating Variance

DataFrame.var() is used to display the variance. It is the average of squared differences from the mean.

```
MarksInformation[['DataMining','AI','Graphics','ImageProcessing','Cryptography']].var()
```

```
DataMining         15.840909
AI                  7.113636
Graphics            9.901515
ImageProcessing     9.969697
Cryptography       11.363636
dtype: float64
```

Calculating Standard Deviation

DataFrame.std() returns the standard deviation of the values. Standard deviation is calculated as the square root of the variance.

```
MarksInformation[['DataMining','AI','Graphics','ImageProcessing','Cryptography']].std()
```

```
DataMining         3.980064
AI                 2.667140
Graphics           3.146667
ImageProcessing    3.157483
Cryptography       3.370999
dtype: float64
```

DataFrame.describe() function displays the descriptive statistical values in a single command. These values help us describe a set of data in a DataFrame.

```
MarksInformation.describe()
```

	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
count	12.000000	12.000000	12.00000	12.000000	12.000000	12.000000
mean	2.000000	19.250000	19.75000	20.416667	21.833333	20.500000
std	0.852803	3.980064	2.66714	3.146667	3.157483	3.370999
min	1.000000	12.000000	15.00000	15.000000	15.000000	13.000000
25%	1.000000	16.500000	18.00000	18.750000	20.750000	19.750000
50%	2.000000	20.500000	19.50000	20.000000	22.500000	21.500000
75%	3.000000	22.250000	21.25000	22.500000	24.000000	23.000000
max	3.000000	24.000000	25.00000	25.000000	25.000000	24.000000

Data Aggregations

Aggregation means to transform the dataset and produce a single numeric value from an array. Aggregation can be applied to one or more columns together. Aggregate functions are max(),min(), sum(), count(), std(), var().

```
MarksInformation.aggregate('max')
```

```
Name                Suci
UnitTest               3
DataMining            24
AI                    25
Graphics              25
ImageProcessing       25
Cryptography          24
dtype: object
```

To use multiple aggregate functions in a single statement

```
MarksInformation.aggregate(['max','count'])
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
max	Suci	3	24	25	25	25	24
count	12	12	12	12	12	12	12

```
MarksInformation['DataMining'].aggregate(['max','min'])
```

```
max    24
min    12
Name: DataMining, dtype: int64
```

We can also use the parameter axis with aggregate function. By default, the value of axis is zero, means columns

Using the above statement with axis=0 gives the same result

```
MarksInformation['DataMining'].aggregate(['max','min'],axis=0)
```

```
max    24
min    12
Name: DataMining, dtype: int64
```

```
MarksInformation[['DataMining','AI']].aggregate('sum',axis=1)
```

```
0     43
1     41
2     33
3     37
4     38
5     40
6     42
7     46
8     37
9     37
10    39
11    35
dtype: int64
```

Sorting a DataFrame

By default, sorting is done in ascending order.

```
MarksInformation.sort_values(by=['Name'])
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
6	Ashravy	1	23	19	20	15	22
7	Ashravy	2	24	22	24	17	21
8	Ashravy	3	12	25	19	21	23
9	Mishti	1	15	22	25	22	22
10	Mishti	2	18	21	25	24	23
11	Mishti	3	17	18	20	25	20
3	Srabanti	1	20	17	22	24	19
4	Srabanti	2	23	15	21	25	15
5	Srabanti	3	22	18	19	23	13
0	Suci	1	22	21	18	20	21
1	Suci	2	21	20	17	22	24
2	Suci	3	14	19	15	24	23

Now, to obtain sorted list of marks scored by all students in Science in Unit Test 2, the following code can be used:

```
test2 = MarksInformation[MarksInformation.UnitTest== 2]
test2
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
1	Suci	2	21	20	17	22	24
4	Srabanti	2	23	15	21	25	15
7	Ashravy	2	24	22	24	17	21
10	Mishti	2	18	21	25	24	23

```
test2.sort_values(by=['DataMining'])
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
10	Mishti	2	18	21	25	24	23
1	Suci	2	21	20	17	22	24
4	Srabanti	2	23	15	21	25	15
7	Ashravy	2	24	22	24	17	21

Write the statement which will sort the marks in English in the DataFrame df based on Unit Test 3, in descending order

```
UnitTest3 =  MarksInformation[MarksInformation.UnitTest == 3]
UnitTest3
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
2	Suci	3	14	19	15	24	23
5	Srabanti	3	22	18	19	23	13
8	Ashravy	3	12	25	19	21	23
11	Mishti	3	17	18	20	25	20

Sort according to descending order of marks in Science

```
UnitTest3.sort_values(by=['AI'],ascending=False)
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
8	Ashravy	3	12	25	19	21	23
2	Suci	3	14	19	15	24	23
5	Srabanti	3	22	18	19	23	13
11	Mishti	3	17	18	20	25	20

A DataFrame can be sorted based on multiple columns. Following is the code of sorting the DataFrame df based on marks in Science in Unit Test 3 in ascending order. If marks in Science are the same, then sorting will be done on the basis of marks in Hindi

Get the data corresponding to marks in Unit Test 3

```
UnitTest3 =  MarksInformation[MarksInformation.UnitTest == 3]
```

Sort the data according to Science and then according to Hindi

```
UnitTest3.sort_values(by=['AI','ImageProcessing'])
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
5	Srabanti	3	22	18	19	23	13
11	Mishti	3	17	18	20	25	20
2	Suci	3	14	19	15	24	23
8	Ashravy	3	12	25	19	21	23

Here, we can see that the list is sorted on the basis of marks in Science. Two students namely, Srabanti and Mishti have equal marks (18) in Science. Therefore for them, sorting is done on the basis of marks in ImageProcessing.

GROUP BY FUNCTIONS

In pandas, DataFrame.GROUP BY() function is used to split the data into groups based on some criteria. Pandas objects like a DataFrame can be split on any of their axes. The GROUP BY function works based on a split-apply-combine strategy which is shown below using a 3-step process:

Step 1: Split the data into groups by creating a GROUP BY object from the original DataFrame.

Step 2: Apply the required function.

Step 3: Combine the results to form a new DataFrame.

Create a GROUP BY Name of the student from DataFrame MarksInformation

```
group=MarksInformation.groupby('Name')
group
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000002489A2FD2B0>
```

Displaying the first entry from each group

```
group.first()
```

Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
Ashravy	1	23	19	20	15	22
Mishti	1	15	22	25	22	22
Srabanti	1	20	17	22	24	19
Suci	1	22	21	18	20	21

```
group.size()
```

```
Name
Ashravy     3
Mishti      3
Srabanti    3
Suci        3
dtype: int64
```

Displaying group data, i.e., group_name, row indexes corresponding to the group and their data type

```
group.groups
```

```
{'Ashravy': [6, 7, 8], 'Mishti': [9, 10, 11], 'Srabanti': [3, 4, 5], 'Suci': [0, 1, 2]}
```

Printing data of a single group

```
group.get_group('Suci')
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	21	18	20	21
1	Suci	2	21	20	17	22	24
2	Suci	3	14	19	15	24	23

Grouping with respect to multiple attributes. Creating a GROUP BY Name and UT

```
group2=MarksInformation.groupby(['Name', 'UnitTest'])
```

```
group2.first()
```

Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
Ashravy	**1**	23	19	20	15	22
	2	24	22	24	17	21
	3	12	25	19	21	23
Mishti	**1**	15	22	25	22	22
	2	18	21	25	24	23
	3	17	18	20	25	20
Srabanti	**1**	20	17	22	24	19
	2	23	15	21	25	15
	3	22	18	19	23	13
Suci	**1**	22	21	18	20	21
	2	21	20	17	22	24
	3	14	19	15	24	23

The above statements show how we create groups by splitting a DataFrame using GROUP BY(). Next step is to apply functions over the groups just created. This is done using Aggregation. Aggregation is a process in which an aggregate function is applied on each group created by

GROUP BY(). It returns a single aggregated statistical value corresponding to each group. It can be used to apply multiple functions over an axis. Be default, functions are applied over columns. Aggregation can be performed using agg() or aggregate() function.

Calculating average marks scored by all students in each subject for each Unit Test

```
group3=MarksInformation.groupby('UnitTest')
```

```
MarksInformation.groupby('UnitTest').agg({'DataMining':'mean','AI':'mean','Graphics':'mean','ImageProcessing':'mean','Cryptography':'mean'})
```

	DataMining	AI	Graphics	ImageProcessing	Cryptography
UnitTest					
1	20.00	19.75	21.25	20.25	21.00
2	21.50	19.50	21.75	22.00	20.75
3	16.25	20.00	18.25	23.25	19.75

Calculate average marks scored in DataMining in each UnitTest

```
MarksInformation.groupby('UnitTest').agg({'DataMining':'mean'})
```

	DataMining
UnitTest	
1	20.00
2	21.50
3	16.25

Write the python statements to print the mean, variance, standard deviation and quartile of the marks scored in DataMining by each student across the UnitTest

```
MarksInformation.groupby('Name').agg({'DataMining':['mean','var','std','quantile']})
```

	DataMining			
	mean	var	std	quantile
Name				
Ashravy	19.666667	44.333333	6.658328	23.0
Mishti	16.666667	2.333333	1.527525	17.0
Srabanti	21.666667	2.333333	1.527525	22.0
Suci	19.000000	19.000000	4.358899	21.0

Altering the Index

We use indexing to access the elements of a DataFrame. It is used for fast retrieval of data. By default, a numeric index starting from 0 is created as a row index, as shown below:

```
MarksInformation
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	21	18	20	21
1	Suci	2	21	20	17	22	24
2	Suci	3	14	19	15	24	23
3	Srabanti	1	20	17	22	24	19
4	Srabanti	2	23	15	21	25	15
5	Srabanti	3	22	18	19	23	13
6	Ashravy	1	23	19	20	15	22
7	Ashravy	2	24	22	24	17	21
8	Ashravy	3	12	25	19	21	23
9	Mishti	1	15	22	25	22	22
10	Mishti	2	18	21	25	24	23
11	Mishti	3	17	18	20	25	20

When we slice the data, we get the original index which is not continuous, e.g. when we select marks of all students in Unit Test 1, we get the following result:

```
UnitTest1 = MarksInformation[MarksInformation.UnitTest == 1]
UnitTest1
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	21	18	20	21
3	Srabanti	1	20	17	22	24	19
6	Ashravy	1	23	19	20	15	22
9	Mishti	1	15	22	25	22	22

Notice that the first column is a non-continuous index since it is slicing of original data. We create a new continuous index alongside this using the reset_index() function, as shown below:

```
UnitTest1 = MarksInformation[MarksInformation.UnitTest == 1]
UnitTest1.reset_index(inplace=True)
UnitTest1
```

	index	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	0	Suci	1	22	21	18	20	21
1	3	Srabanti	1	20	17	22	24	19
2	6	Ashravy	1	23	19	20	15	22
3	9	Mishti	1	15	22	25	22	22

We can change the index to some other column of the data

```
UnitTest1.set_index('Name',inplace=True)
UnitTest1
```

	index	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
Name							
Suci	0	1	22	21	18	20	21
Srabanti	3	1	20	17	22	24	19
Ashravy	6	1	23	19	20	15	22
Mishti	9	1	15	22	25	22	22

We can revert back to previous index by using following statement:

```
UnitTest1.reset_index('Name', inplace = True)
UnitTest1
```

	Name	index	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	0	1	22	21	18	20	21
1	Srabanti	3	1	20	17	22	24	19
2	Ashravy	6	1	23	19	20	15	22
3	Mishti	9	1	15	22	25	22	22

Other DataFrame Operations

1.Reshaping Data

(A) Pivot

The pivot function is used to reshape and create a new DataFrame from the original one. Consider the following example of sales and profit data of four stores: Boi Bichitra,Pathak Shamabesh,Puthighar and Muktodhara for the years 2016, 2017 and 2018.

```
data={'Store':['Boi Bichitra','Pathak Shamabesh','Puthighar','Boi Bichitra','Muktodhara','Puthighar','Boi Bichitra','Muktodhara','Puthighar'
      'Year':[2016,2016,2016,2017,2017,2017,2018,2018,2018],
      'Total_sales(TK)':[12000,330000,420000,20000,10000,450000,30000, 11000,89000],
      'Total_profit(TK)':[1100,5500,21000,32000,9000,45000,3000,1900,23000]
}
```

```
Transaction=pd.DataFrame(data)
Transaction
```

	Store	Year	Total_sales(TK)	Total_profit(TK)
0	Boi Bichitra	2016	12000	1100
1	Pathak Shamabesh	2016	330000	5500
2	Puthighar	2016	420000	21000
3	Boi Bichitra	2017	20000	32000
4	Muktodhara	2017	10000	9000
5	Puthighar	2017	450000	45000
6	Boi Bichitra	2018	30000	3000
7	Muktodhara	2018	11000	1900
8	Puthighar	2018	89000	23000

Let us try to answer the following queries on the above data.

1) What was the total sale of store Boi Bichitra in all the years? Python statements to perform this task will be as follows:

```
Store1 = Transaction[Transaction.Store=='Boi Bichitra']
Store1
```

	Store	Year	Total_sales(TK)	Total_profit(TK)
0	Boi Bichitra	2016	12000	1100
3	Boi Bichitra	2017	20000	32000
6	Boi Bichitra	2018	30000	3000

```
Store1['Total_sales(TK)'].sum()
```

```
62000
```

2) What is the maximum sale value by store Puthighar in any year?

```
Store3 = Transaction[Transaction.Store=='Puthighar']
Store3
```

	Store	Year	Total_sales(TK)	Total_profit(TK)
2	Puthighar	2016	420000	21000
5	Puthighar	2017	450000	45000
8	Puthighar	2018	89000	23000

```
Store3['Total_sales(TK)'].max()
```

```
450000
```

3) Which store had the maximum total sale in all the years?

```
Store1= Transaction[Transaction.Store=='Boi Bichitra']
Store1
```

	Store	Year	Total_sales(TK)	Total_profit(TK)
0	Boi Bichitra	2016	12000	1100
3	Boi Bichitra	2017	20000	32000
6	Boi Bichitra	2018	30000	3000

```
Store2=Transaction[Transaction.Store=='Muktodhara']
Store2
```

	Store	Year	Total_sales(TK)	Total_profit(TK)
4	Muktodhara	2017	10000	9000
7	Muktodhara	2018	11000	1900

```
Store3 = Transaction[Transaction.Store=='Puthighar']
Store3
```

	Store	Year	Total_sales(TK)	Total_profit(TK)
2	Puthighar	2016	420000	21000
5	Puthighar	2017	450000	45000
8	Puthighar	2018	89000	23000

```
Store4 = Transaction[Transaction.Store=='Pathak Shamabesh']
Store4
```

	Store	Year	Total_sales(TK)	Total_profit(TK)
1	Pathak Shamabesh	2016	330000	5500

```
Store1Total = Store1['Total_sales(TK)'].sum()
Store1Total
```

```
62000
```

```
Store2Total = Store2['Total_sales(TK)'].sum()
Store2Total
```

```
21000
```

```
Store3Total = Store3['Total_sales(TK)'].sum()
Store3Total
```

```
959000
```

```
Store4Total = Store4['Total_sales(TK)'].sum()
Store4Total
```

```
330000
```

```
max(Store1Total,Store2Total,Store3Total,Store4Total)
```

```
959000
```

Notice that we have to slice the data corresponding to a particular store and then answer the query. Now, let us reshape the data using pivot and see the difference.

```
pivot1=Transaction.pivot(index='Store',columns='Year',values='Total_sales(TK)')
pivot1
```

Year	2016	2017	2018
Store			
Boi Bichitra	12000.0	20000.0	30000.0
Muktodhara	NaN	10000.0	11000.0
Pathak Shamabesh	330000.0	NaN	NaN
Puthighar	420000.0	450000.0	89000.0

As can be seen above, the value of Total_sales (Rs) for every row in the original table has been transferred to the new table: pivot1, where each row has data of a store and each column has data of a year. Those cells in the new pivot table which do not have a matching entry in the original one are filled with NaN. For instance, we did not have values corresponding to sales of Store S2 in 2016, thus the appropriate cell in pivot1 is filled with NaN.

1) What was the total sale of store Boi Bichitra in all the years?

```
pivot1.loc['Boi Bichitra'].sum()
```

```
62000.0
```

2) What is the maximum sale value by store Puthighar in any year?

```
pivot1.loc['Puthighar'].max()
```

```
450000.0
```

3.Which store had the maximum total sale?

```
Store1Total = pivot1.loc['Boi Bichitra'].sum()
Store1Total
```

```
62000.0
```

```
Store2Total = pivot1.loc['Muktodhara'].sum()
Store2Total
```

```
21000.0
```

```
Store3Total = pivot1.loc['Puthighar'].sum()
Store3Total
```

```
959000.0
```

```
Store4Total = pivot1.loc['Pathak Shamabesh'].sum()
Store4Total
```

```
330000.0
```

```
max(Store1Total,Store2Total,Store3Total,Store4Total)
```

```
959000.0
```

(B) Pivoting by Multiple Columns

```
pivot2=Transaction.pivot(index='Store',columns='Year',values=['Total_sales(TK)','Total_profit(TK)'])
pivot2
```

	Total_sales(TK)			Total_profit(TK)		
Year	2016	2017	2018	2016	2017	2018
Store						
Boi Bichitra	12000.0	20000.0	30000.0	1100.0	32000.0	3000.0
Muktodhara	NaN	10000.0	11000.0	NaN	9000.0	1900.0
Pathak Shamabesh	330000.0	NaN	NaN	5500.0	NaN	NaN
Puthighar	420000.0	450000.0	89000.0	21000.0	45000.0	23000.0

Let us consider another example, where suppose we have stock data corresponding to a store as:

```
data={'Item':['Pen','Pen','Pencil','Pencil'
,'Pen','Pen'],
'Color':['Red','Red','Black','Black','Blue','Blue'],
'Price(TK)':[10,25,7,5,50,20],
'Units_in_stock':[50,10,47,34,55,14]
}
```

```
Stock=pd.DataFrame(data)
Stock
```

	Item	Color	Price(TK)	Units_in_stock
0	Pen	Red	10	50
1	Pen	Red	25	10
2	Pencil	Black	7	47
3	Pencil	Black	5	34
4	Pen	Blue	50	55
5	Pen	Blue	20	14

Now, let us assume, we have to reshape the above table with Item as the index and Color as the column. We will use pivot function as given below:

pivot3=Stock.pivot(index='Item',columns='Color',values='Units_in_stock')

But this statement results in an error: ValueError: Index contains duplicate entries, cannot reshape. This is because duplicate data can't be reshaped using pivot function. Hence, before calling the pivot() function, we need to ensure that our data do not have rows with duplicate values for the specified columns. If we can't ensure this, we may have to use pivot_table function instead.

(C) Pivot Table

It works like a pivot function, but aggregates the values from rows with duplicate entries for the specified columns. In other words, we can use aggregate functions like min, max, mean etc, wherever we have duplicate entries. The default aggregate function is mean.

```
Stock1 = Stock.pivot_table(index=['Item','Color'])
Stock1
```

		Price(TK)	Units_in_stock
Item	**Color**		
Pen	**Blue**	35.0	34.5
	Red	17.5	30.0
Pencil	**Black**	6.0	40.5

```
import warnings
warnings.filterwarnings('ignore')
```

Mean has been used as the default aggregate function. Price of the blue pen in the original data is 50 and 20. Mean has been used as aggregate and the price of the blue pen is 35 in df1. We can use multiple aggregate functions on the data. Below example shows the use of the sum, max and np.mean function.

```
import numpy as np
pivot_table1=Stock.pivot_table(index='Item',columns='Color',values='Units_in_stock',aggfunc=[sum,max,np.mean])
pivot_table1
```

	sum			max			mean		
Color	**Black**	**Blue**	**Red**	**Black**	**Blue**	**Red**	**Black**	**Blue**	**Red**
Item									
Pen	NaN	69.0	60.0	NaN	55.0	50.0	NaN	34.5	30.0
Pencil	81.0	NaN	NaN	47.0	NaN	NaN	40.5	NaN	NaN

Pivoting can also be done on multiple columns. Further, different aggregate functions can be applied on different columns. The following example demonstrates pivoting on two columns - Price(Rs) and Units_in_stock. Also, the application of len() function on the column Price(Rs) and mean() function of column Units_in_ stock is shown in the example. Note that the aggregate function len returns the number of rows corresponding to that entry.

```
pivot_table2=Stock.pivot_table(index='Item',columns='Color',values=['Price(TK)','Units_in_stock'],aggfunc={"Price(TK)":len,"Units_in_stock":
pivot_table2
```

	Price(TK)			Units_in_stock		
Color	**Black**	**Blue**	**Red**	**Black**	**Blue**	**Red**
Item						
Pen	NaN	2.0	2.0	NaN	34.5	30.0
Pencil	2.0	NaN	NaN	40.5	NaN	NaN

Write the statement to print the maximum price of pen of each color.

```
pen=Stock[Stock.Item=='Pen']
pen
```

	Item	Color	Price(TK)	Units_in_stock
0	Pen	Red	10	50
1	Pen	Red	25	10
4	Pen	Blue	50	55
5	Pen	Blue	20	14

```
redpen=pen.pivot_table(index='Item',columns=['Color'],values=['Price(TK)'],aggfunc=[max])
redpen
```

	max	
	Price(TK)	
Color	**Blue**	**Red**
Item		
Pen	50	25

Handling Missing Values

Missing values create a lot of problems during data analysis and have to be handled properly. The two most common strategies for handling missing values explained in this section are: i) drop the object having missing values, ii) fill or estimate the missing value

1.Checking Missing Values

```
 marks = {
'Name':['Suci','Suci','Suci','Srabanti','Srabanti','Srabanti', 'Ashravy','Ashravy','Ashravy','Mishti','Mishti','Mishti'],
 'UnitTest':[1,2,3,1,2,3,1,2,3,1,2,3],
 'DataMining':[22,21,14,20,23,22,23,24,12,15,18,17],
 'AI':[20,np.NaN,19,17,15,18,19,22,np.NaN,22,21,18],
 'Graphics':[18,17,15,22,21,19,20,24,19,25,25,20],
 'ImageProcessing':[20,22,24,24,25,23,15,np.NaN,21,22,24,25],
 'Cryptography':[24,np.NaN,23,19,15,13,22,21,23,22,23,np.NaN] }
Marks = pd.DataFrame(marks)
Marks
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	20.0	18	20.0	24.0
1	Suci	2	21	NaN	17	22.0	NaN
2	Suci	3	14	19.0	15	24.0	23.0
3	Srabanti	1	20	17.0	22	24.0	19.0
4	Srabanti	2	23	15.0	21	25.0	15.0
5	Srabanti	3	22	18.0	19	23.0	13.0
6	Ashravy	1	23	19.0	20	15.0	22.0
7	Ashravy	2	24	22.0	24	NaN	21.0
8	Ashravy	3	12	NaN	19	21.0	23.0
9	Mishti	1	15	22.0	25	22.0	22.0
10	Mishti	2	18	21.0	25	24.0	23.0
11	Mishti	3	17	18.0	20	25.0	NaN

```
Marks.isnull()
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	False	False	False	False	False	False	False
1	False	False	False	True	False	False	True
2	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False
5	False	False	False	False	False	False	False
6	False	False	False	False	False	False	False
7	False	False	False	False	False	True	False
8	False	False	False	True	False	False	False
9	False	False	False	False	False	False	False
10	False	False	False	False	False	False	False
11	False	False	False	False	False	False	True

One can check for each individual attribute also, e.g. the following statement checks whether attribute 'AI' has a missing value or not. It returns True for each row where there is a missing value for attribute 'AI', and False otherwise.

```
Marks['AI'].isnull()
```

```
0     False
1      True
2     False
3     False
4     False
5     False
6     False
7     False
8      True
9     False
10    False
11    False
Name: AI, dtype: bool
```

To check whether a column (attribute) has a missing value in the entire dataset, any() function is used. It returns True in case of missing value else returns False.

```
Marks['AI'].isnull().any()
```

```
True
```

```
Marks['DataMining'].isnull().any()
```

```
False
```

To find the number of NaN values corresponding to each attribute, one can use the sum() function along with isnull() function, as shown below:

```
Marks.isnull().sum()
```

```
Name               0
UnitTest           0
DataMining         0
AI                 2
Graphics           0
ImageProcessing    1
Cryptography       2
dtype: int64
```

To find the total number of NaN in the whole dataset, one can use df.isnull().sum().sum().

```
Marks.isnull().sum().sum()
```

```
5
```

Write a program to find the percentage of marks scored by Suci in DataMining.

```
Suci = Marks[Marks['Name']=='Suci']
print('Marks Scored by Suci')
Suci
```

```
Marks Scored by Suci
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	20.0	18	20.0	24.0
1	Suci	2	21	NaN	17	22.0	NaN
2	Suci	3	14	19.0	15	24.0	23.0

```
DataMining = Suci['DataMining']
print("Marks Scored by Suci in DataMining")
DataMining
```

```
Marks Scored by Suci in DataMining
0    22
1    21
2    14
Name: DataMining, dtype: int64
```

```
row = len(DataMining) # Number of Unit Testsheld. Here row will be 4
```

```
row
```

```
3
```

```
print("Percentage of Marks Scored by Suci in DataMining\n\n",(DataMining.sum()*100)/(25*row),"%")
```

```
Percentage of Marks Scored by Suci in DataMining

 76.0 %
```

Write a python program to find the percentage of marks obtained by Suci in Cryptography subject.

```
AI = Suci['AI']
print("Marks Scored by Suci in Cryptography")
AI
```

```
Marks Scored by Suci in Cryptography
0    20.0
1     NaN
2    19.0
Name: AI, dtype: float64
```

```
row = len(AI) # here, row will be 4,the number of Unit Tests
row
```

```
3
```

```
print("Percentage of Marks Scored by Suci in AI\n\n", AI.sum()*100/(25*row),"%")
```

```
Percentage of Marks Scored by Suci in AI

 52.0 %
```

Here, notice that Suci was absent in Unit Test 4 in AI Subject. While computing the percentage, marks of the fourth test have been considered as 0.

2.Dropping Missing Values

```
Marks1 = Marks.dropna()
Marks1
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	20.0	18	20.0	24.0
2	Suci	3	14	19.0	15	24.0	23.0
3	Srabanti	1	20	17.0	22	24.0	19.0
4	Srabanti	2	23	15.0	21	25.0	15.0
5	Srabanti	3	22	18.0	19	23.0	13.0
6	Ashravy	1	23	19.0	20	15.0	22.0
9	Mishti	1	15	22.0	25	22.0	22.0
10	Mishti	2	18	21.0	25	24.0	23.0

Now, let us consider the following code:

```
Suci=Marks[Marks.Name=='Suci']
Suci
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	20.0	18	20.0	24.0
1	Suci	2	21	NaN	17	22.0	NaN
2	Suci	3	14	19.0	15	24.0	23.0

```
AI = Suci['AI']
```

```
print("\nMarks Scored by Suci in AI\n")
AI
```

```
Marks Scored by Suci in AI

0    20.0
1     NaN
2    19.0
Name: AI, dtype: float64
```

```
row = len(AI)
print("\nPercentage of Marks Scored by Suci in AI\n")
print(AI.sum()*100/(25*row),"%")
```

```
Percentage of Marks Scored by Suci in AI

52.0 %
```

```
Suci1=Suci.dropna(axis=0)
```

```
AI = Suci1['AI']
```

```
row = len(AI)
print("\nPercentage of Marks Scored by Suci in AI\n")
print(AI.sum()*100/(25*row),"%")
```

```
Percentage of Marks Scored by Suci in AI

78.0 %
```

3.Estimating Missing Values

Missing values can be filled by using estimations or approximations e.g a value just before (or after) the missing value, average/minimum/maximum of the values of that attribute, etc. In some cases, missing values are replaced by zeros (or ones). The fillna(num) function can be used to replace missing value(s) by the value specified in num. For example, fillna(0) replaces missing value by 0. Similarly fillna(1) replaces missing value by 1. Following code replaces missing values by 0 and computes the percentage of marks scored by Raman in Science.

```
#Marks Scored by Suci in all the subjects across the tests
Suci = Marks.loc[Marks['Name']=='Suci']
Suci
```

	Name	UnitTest	DataMining	AI	Graphics	ImageProcessing	Cryptography
0	Suci	1	22	20.0	18	20.0	24.0
1	Suci	2	21	NaN	17	22.0	NaN
2	Suci	3	14	19.0	15	24.0	23.0

```
(row,col) =Suci.shape
(row,col)
```

```
(3, 7)
```

```
AI = Suci.loc[:,'AI']
print("Marks Scored by Suci in AI")
AI
```

```
Marks Scored by Suci in AI
0    20.0
1     NaN
2    19.0
Name: AI, dtype: float64
```

```
FillZeroAI = AI.fillna(0)
print('\nMarks Scored by Suci in AI with Missing Values Replaced with Zero')
FillZeroAI
```

```

Marks Scored by Suci in AI with Missing Values Replaced with Zero
0    20.0
1     0.0
2    19.0
Name: AI, dtype: float64
```

```
print("Percentage of Marks Scored by Suci in AI\n\n",FillZeroAI.sum()*100/(25*row),"%")
```

```
Percentage of Marks Scored by Suci in AI

 52.0 %
```

df.fillna(method='pad') replaces the missing value by the value before the missing value while df.fillna(method='bfill') replaces the missing value by the value after the missing value. Following code replaces the missing value of Cryptography then computes the percentage of marks obtained by Suci.

```
Cryptography = Suci.loc[:,'Cryptography']
print("Marks Scored by Suci in Cryptography")
Cryptography
```

```
Marks Scored by Suci in Cryptography
0    24.0
1     NaN
2    23.0
Name: Cryptography, dtype: float64
```

```
FillPadCryptography = Cryptography.fillna(method='pad')
print('\nMarks Scored by Suci in Cryptography with Missing Values Replaced by Previous TestMarks')
FillPadCryptography
```

```
Marks Scored by Suci in Cryptography with Missing Values Replaced by Previous TestMarks
0    24.0
1    24.0
2    23.0
Name: Cryptography, dtype: float64
```

```
row = len(Cryptography)
```

```
print("Percentage of Marks Scored by Suci in Cryptography")
print(FillPadCryptography.sum()*100/(25*row),"%")
```

```
Percentage of Marks Scored by Suci in Cryptography
94.66666666666667 %
```

EXERCISE-1

a.) To create the DataFrame for the given table:

```
import pandas as pd


data = {
    'Item': ['TV', 'TV', 'TV', 'AC'],
    'Company': ['LG', 'VIDEOCON', 'LG', 'SONY'],
    'Rupees': [12000, 10000, 15000, 14000],
    'USD': [700, 650, 800, 750]
}

Product = pd.DataFrame(data)
print("Initial DataFrame:")
Product
```

Initial DataFrame:

	Item	Company	Rupees	USD
0	TV	LG	12000	700
1	TV	VIDEOCON	10000	650
2	TV	LG	15000	800
3	AC	SONY	14000	750

b) To add new rows in the DataFrame:

```
new_data = {
    'Item': ['TV', 'AC'],
    'Company': ['SAMSUNG', 'LG'],
    'Rupees': [13000, 16000],
    'USD': [720, 900]
}

new_rows = pd.DataFrame(new_data)

# Append the new rows
Product= pd.concat([Product,new_rows], ignore_index=True)
print("\nDataFrame after adding new rows:")
Product
```

DataFrame after adding new rows:

	Item	Company	Rupees	USD
0	TV	LG	12000	700
1	TV	VIDEOCON	10000	650
2	TV	LG	15000	800
3	AC	SONY	14000	750
4	TV	SAMSUNG	13000	720
5	AC	LG	16000	900

c) To display the maximum price of LG TV:

```
maxPrice_lg_tv = Product[(Product['Item'] == 'TV') & (Product['Company'] == 'LG')]['Rupees'].max()
print(f"\nMaximum price of LG TV: {maxPrice_lg_tv}")
```

```
Maximum price of LG TV: 15000
```

d) To display the sum of all products:

```
totalSum = Product['Rupees'].sum()
print(f"\nSum of all products: {totalSum} Rupees")
```

```
Sum of all products: 80000 Rupees
```

e) To display the median of the USD of Sony products:

```
medianSonyUsd = Product[Product['Company'] == 'SONY']['USD'].median()
print(f"\nMedian of USD for Sony products: {medianSonyUsd}")
```

```
Median of USD for Sony products: 750.0
```

EXERCISE-2

a) To create the DataFrame:

```
import numpy as np
data={
    'Name':['Aparna','Pankaj','Ram','Ramesh','Naveen','Krishnav','Brauma'],
     'Degree':['MBA','BCA','M.Tech','MBA',np.NaN,'BCA','MBA'],
      'Score':[90.0,np.NaN,80,98,97,78,89]
}
Mark=pd.DataFrame(data)
Mark
```

	Name	Degree	Score
0	Aparna	MBA	90.0
1	Pankaj	BCA	NaN
2	Ram	M.Tech	80.0
3	Ramesh	MBA	98.0
4	Naveen	NaN	97.0
5	Krishnav	BCA	78.0
6	Brauma	MBA	89.0

b) To print the Degree and maximum marks in each stream:

```
maxMarks = Mark.groupby('Degree')['Score'].max()
print("\nMaximum marks in each stream:")
maxMarks
```

```
Maximum marks in each stream:
Degree
BCA       78.0
M.Tech    80.0
MBA       98.0
Name: Score, dtype: float64
```

c) To fill the NaN with 76:

```
Mark_filled = Mark.fillna(76)
print("\nDataFrame after filling NaN with 76:")
Mark_filled
```

```
DataFrame after filling NaN with 76:
```

	Name	Degree	Score
0	Aparna	MBA	90.0
1	Pankaj	BCA	76.0
2	Ram	M.Tech	80.0
3	Ramesh	MBA	98.0
4	Naveen	76	97.0
5	Krishnav	BCA	78.0
6	Brauma	MBA	89.0

d) To set the index to Name:

```
Mark_indexed = Mark_filled.set_index('Name')
print("\nDataFrame with Name as index:")
Mark_indexed
```

```
DataFrame with Name as index:
```

Name	Degree	Score
Aparna	MBA	90.0
Pankaj	BCA	76.0
Ram	M.Tech	80.0
Ramesh	MBA	98.0
Naveen	76	97.0
Krishnav	BCA	78.0
Brauma	MBA	89.0

e) To display the name and degree-wise average marks of each student:

```
average_marks = Mark_filled.groupby(['Name', 'Degree'])['Score'].mean().reset_index()
print("\nName and Degree wise average marks of each student:")
average_marks
```

```
Name and Degree wise average marks of each student:
```

	Name	Degree	Score
0	Aparna	MBA	90.0
1	Brauma	MBA	89.0
2	Krishnav	BCA	78.0
3	Naveen	76	97.0
4	Pankaj	BCA	76.0
5	Ram	M.Tech	80.0
6	Ramesh	MBA	98.0

f) To count the number of students in MBA:

```
MBAcount = Mark[Mark['Degree'] == 'MBA'].shape[0]
print(f"\nNumber of students in MBA: {MBAcount}")
```

```
Number of students in MBA: 3
```

g) To print the mode marks for BCA students:

```
BCAmode = Mark[Mark['Degree'] == 'BCA']['Score'].mode()[0]
print(f"\nMode marks for BCA students: {BCAmode}")
```

```
Mode marks for BCA students: 78.0
```

END OF DataFrame Assignment

Series

A Series is a one-dimensional array that holds a sequence of values, which can be of any data type (such as int, float, list, or string). By default, these values are labeled with numeric indices starting at zero. The label linked to each value is referred to as its index. It's also possible to use other data types for the index. We can think of a Pandas Series as similar to a column in a spreadsheet. Example of a series containing names of flowers is given below:

Index	Values
0	Rose
1	Sunflower
2	Jasmine
3	Daisy
4	Lily
5	Tulip
6	Lavender
7	Orchid
8	Daffodil

Creation of **Series**

To create or work with series in Pandas, the first step is to import the Pandas library. There are various methods available in Pandas library to create and work with series.

Creation of Series from **Scalar** Values

We can create a Series using scalar values, as demonstrated in the example below:

```
import pandas as pd  #import Pandas with alias pd

First_series = pd.Series([1,20,300,4000,50000]) #create a Series
print(First_series) #Display the series
```

```
0        1
1       20
2      300
3     4000
4    50000
dtype: int64
```

Notice that the output is displayed in two columns: the **index** on the left and the **data values** on the right. If we do not explicitly define an index when creating a series, the default indices will range from 0 to N-1, where N represents the total number of data elements.

Additionally, we can assign custom labels to the index and use them to access elements within the Series. The example below demonstrates this with a numeric index arranged in random order-

```
Second_series = pd.Series(["Mango","Blueberry","Apple","Pear","Avocado"],index=[2,4,1,3,5])

print(Second_series) #Display the series
```

```
2        Mango
4    Blueberry
1        Apple
3         Pear
5      Avocado
dtype: object
```

We can also use letters or strings as indices, such as in the following example:

```
Third_series = pd.Series([3,5,7],index=["Three","Five","Seven"])

print(Third_series)
```

```
Three    3
Five     5
Seven    7
dtype: int64
```

Here, data values 3,5,7 have index values Three, Five and Seven respectively.

Creation of Series from **NumPy Arrays**

We can create a series from a one-dimensional NumPy array, can be shown as:

```
import numpy as np # import NumPy with alias np
import pandas as pd

First_array = np.array([10,20,30,40,50,60,70,80,90,100])
Fourth_series = pd.Series(First_array)

print(Fourth_series)
```

```
0     10
1     20
2     30
3     40
4     50
5     60
6     70
7     80
8     90
9    100
dtype: int32
```

The example below demonstrates that letters or strings can be used as indices:

```
Fifth_series = pd.Series(First_array , index = ["Ten","Twenty", "Thirty", "Forty","Fifty","Sixty","Seventy", "Eighty","Ninety","One hundred"

print(Fifth_series)
```

```
Ten             10
Twenty          20
Thirty          30
Forty           40
Fifty           50
Sixty           60
Seventy         70
Eighty          80
Ninety          90
One hundred    100
dtype: int32
```

When index labels are provided along with an array, the length of the index and the array must match; otherwise, a ***ValueError*** will occur. In the example below, the `First_array` has 10 values, but only 7 indices are specified, leading to a **ValueError**.

```
#Sixth_series = pd.Series(First_array , index = ["Ten","Twenty", "Thirty", "Forty","Fifty","Sixty","Seventy"])
```

So, we have to be careful while specifying index in respect of the data values.

Creation of Series from **Dictionary**

Remember that a Python dictionary contains ***key-value*** pairs, allowing quick retrieval of a value when its key is known. These dictionary keys can be used to create an index for a Series. In the example below, the keys from the dictionary `first_dictionary` become the indices in the Series.

```
first_dictionary = {'Fruit': 'Mango',
                    'Flower':'Lavender',
                    'Vegetable': 'Tomato'}

print(first_dictionary)
```

```
{'Fruit': 'Mango', 'Flower': 'Lavender', 'Vegetable': 'Tomato'}
```

```
seventh_series = pd.Series(first_dictionary)
print(seventh_series)
```

```
Fruit           Mango
Flower       Lavender
Vegetable      Tomato
dtype: object
```

Accessing Elements of a Series

The two main methods for accessing elements in a Series. Those are-

- Indexing;
- Slicing.

Indexing

Indexing in a Series is similar to that in NumPy arrays and is used to access elements within a Series. There are two types of indexes: ***positional*** and ***labeled***.

A positional index uses an integer corresponding to the element's position in the Series, starting from 0. In contrast, a labeled index uses a custom label defined by the user.

Here is an example that shows usage of the positional index for accessing a value from a Series-

```
eigth_series = pd.Series([1,20,300,4000,50000])
eigth_series[3]
```

```
4000
```

Here, the value 4000 is displayed for the positional index 3.

When labels are specified, we can use labels as indices while selecting values from a Series, as shown below. Here, the value 10 is displayed for the labelled index Ten.

```
ninth_series = pd.Series([2,4,6,8,10,12],index=["Two","Four","Six","Eight","Ten","Twelve"])
ninth_series["Ten"]
```

```
10
```

In the example below, value Fruit is displayed for the labelled index Mango.

```
tenth_series = pd.Series(['Flower', 'Fruit', 'Color', 'Vegetable'],index=['Rose', 'Mango', 'Green', 'Tomato'])

tenth_series['Mango']
```

```
'Fruit'
```

```
import warnings
warnings.filterwarnings('ignore')
```

We can also access an element of the series using the positional index:

```
tenth_series[3]
```

```
'Vegetable'
```

More than one element of a series can be accessed using a list of positional integers or a list of index labels as shown in the following examples:

```
tenth_series[[1,2]]
```

```
Mango    Fruit
Green    Color
dtype: object
```

We can modify the index values of a series by assigning new ones, as demonstrated in the example below:

```
tenth_series.index=[1,2,3,4]
tenth_series
```

```
1       Flower
2        Fruit
3        Color
4    Vegetable
dtype: object
```

Slicing

Occasionally, it might be necessary to retrieve a portion of a series, which can be accomplished through ***slicing***. This process is akin to slicing with NumPy arrays. WE can specify the desired segment of the series by defining the start and end parameters [start:end] with the series name. When using positional indices for slicing, the value at the end index is not included, meaning only (end - start) number of data values are extracted from the series. Let's see the following example-

```
eleventh_series= pd.Series(['Tulip', 'Cheery', 'Yellow', 'Potato'],index=['Flower', 'Fruit', 'Color', 'Vegetable'])
eleventh_series[0:3]
```

```
Flower     Tulip
Fruit     Cheery
Color     Yellow
dtype: object
```

As we can see that in the above output, only data values at indices 0, 1 and 2 are displayed. If labelled indexes are used for slicing, then value at the end index label is also included in the output, as example:

```
eleventh_series['Flower':'Color']
```

```
Flower     Tulip
Fruit     Cheery
Color     Yellow
dtype: object
```

We can also get the series in reverse order, as example:

```
eleventh_series[ : : -1]
```

```
Vegetable    Potato
Color        Yellow
Fruit        Cheery
```

```
Flower        Tulip
dtype: object
```

We can also use slicing to modify the values of series elements as shown in the following example:

```
import numpy as np

color_series = pd.Series(np.arange(3,8,1),index = ['Three', 'Five', 'Seven', 'Nine', 'Eleven'])
color_series
```

```
Three     3
Five      4
Seven     5
Nine      6
Eleven    7
dtype: int32
```

```
color_series[1]= 5
color_series[2]= 7
color_series[3]= 9
color_series[4]= 11
color_series
```

```
Three      3
Five       5
Seven      7
Nine       9
Eleven    11
dtype: int32
```

When updating values in a series with slicing, the value at the end index is not included. However, if we perform slicing using labels, the value at the end index label will be updated.

```
color_series['Five':'Nine']= 579
color_series
```

```
Three       3
Five      579
Seven     579
Nine      579
Eleven     11
dtype: int32
```

So these were some ways of **Accessing Elements** of a Series

Attributes of Series

We can access specific properties, known as attributes, of a series by referring to those attributes with the series name. Here for example we are using a series of flowers.

```
Flower_series= pd.Series(['Rose','Jasmine','Lavender', 'Sunflower','Lily','Orchid','Daisy'])
print(Flower_series)
```

```
0         Rose
1      Jasmine
2     Lavender
3    Sunflower
4         Lily
5       Orchid
6        Daisy
dtype: object
```

Attribute: name

Assigns a name to the Series

```
# Attribute: name
# assigns a name to the Series

Flower_series.name = 'Flowers'
print(Flower_series)
```

```
0         Rose
1      Jasmine
2     Lavender
3    Sunflower
4         Lily
5       Orchid
6        Daisy
Name: Flowers, dtype: object
```

Attribute: index.name

Gives a name to the series' index

```
# Attribute: index.name
# agives a name to the series' index

Flower_series.index.name = 'Flowers'
print(Flower_series)
```

```
Flowers
0         Rose
1      Jasmine
2     Lavender
3    Sunflower
4         Lily
5       Orchid
6        Daisy
Name: Flowers, dtype: object
```

Attribute: values

Displays a list of the values in the series

```
# Attribute: values
# displays a list of the values in the series

print(Flower_series.values)
```

```
['Rose' 'Jasmine' 'Lavender' 'Sunflower' 'Lily' 'Orchid' 'Daisy']
```

Attribute: size

Shows the count of values in the Series object

```
# Attribute: size
# shows the count of values in the Series object

print(Flower_series.size)
```

```
7
```

Attribute: empty

Displays True if the series is empty, and False if it contains elements

```
# Attribute: empty
# displays True if the series is empty, and False if it contains elements

Flower_series.empty
```

```
False
```

```
empty_series= pd.Series()
empty_series.empty
```

```
True
```

Methodes of Series

Let's explore various methods available for Pandas Series. Let's examine the following series:

```
evenNumber_series = pd.Series([2,4,6,8,10,12,14,16],index=["Two","Four","Six","Eight","Ten","Twelve","Fourteen", "Sixteen"])
print(evenNumber_series)
```

```
Two          2
Four         4
Six          6
Eight        8
Ten         10
Twelve      12
Fourteen    14
Sixteen     16
dtype: int64
```

Methode: head(n)

Returns the first n elements of the series. If n is not provided, it defaults to 5, displaying the first five elements.

```
# Methose:  head(n)
# Returns the first n elements of the series. If n is not provided, it defaults to 5, displaying the first five elements.

evenNumber_series.head(4)
```

```
Two      2
Four     4
Six      6
Eight    8
dtype: int64
```

```
evenNumber_series.head() #by default the quantity of elements are 5. So first 5 elements are shown
```

```
Two       2
Four      4
Six       6
Eight     8
Ten      10
dtype: int64
```

Methode: count()

Returns the number of non-NaN values in the Series

```
# Methode: count()
# Returns the number of non-NaN values in the Series

evenNumber_series.count()
```

```
8
```

Methode: tail(n)

Returns the last n elements of the series. If n is not specified, it defaults to 5, showing the last five elements.

```
# Methose:  tail(n)
# Returns the last n elements of the series. If n is not specified, it defaults to 5, showing the last five elements.

evenNumber_series.tail(4)
```

```
Ten         10
Twelve      12
Fourteen    14
Sixteen     16
dtype: int64
```

```
evenNumber_series.tail() #by default the quantity of elements are 5. So last 5 elements are shown
```

```
Eight        8
Ten         10
Twelve      12
Fourteen    14
Sixteen     16
dtype: int64
```

Mathematical Operations on Series

When performing mathematical operations on series, indices are matched, and any missing values are automatically filled with NaN. To understand mathematical operations on series in Pandas, consider the following examples with series_num1 and series_num2-

```
series_num1 = pd.Series([-1,2,0,4,-3], index = ['a', 'e', 'i', 'o', 'u'])

series_num1
```

```
a   -1
e    2
i    0
o    4
u   -3
dtype: int64
```

```
series_num2 = pd.Series([11,22,33,44,55,66,77], index = ['a','b', 'c', 'd','e', 'f', 'g'])
series_num2
```

```
a    11
b    22
c    33
d    44
e    55
f    66
g    77
dtype: int64
```

Addition of two Series

This can be achieved in **two** ways. The first method involves directly adding the two series together, as demonstrated in the following code.

Note: the output of addition is NaN if one of the elements or both elements have no value.

```
series_num1 + series_num2
```

```
a    10.0
b     NaN
c     NaN
d     NaN
```

```
e    57.0
f     NaN
g     NaN
i     NaN
o     NaN
u     NaN
dtype: float64
```

Here we can see values of index- b, c, d, f, g, i, o, u are NaN because the output of addition is NaN if one of the elements or both elements have no value.

The **second** method is used when we want to avoid NaN values in the output.

By using the `add()` method along with the `fill_value` parameter, we can replace missing values with a specified value. Calling `seriesA.add(seriesB)` works the same as `series_num1 + series_num2`, but the `add()` method allows us to explicitly set a fill value for any missing elements in series_num1 or series_num2.

```
series_num1.add(series_num2, fill_value=0)
```

```
a    10.0
b    22.0
c    33.0
d    44.0
e    57.0
f    66.0
g    77.0
i     0.0
o     4.0
u    -3.0
dtype: float64
```

This is how addition can be done in series.

Similarly to addition- subtraction, multiplication, and division can be performed using the respective mathematical operators or by explicitly calling the relevant methods.

Subtraction of two Series

As mentioned previously, it can be done in *two* different ways, as shown in the following examples:

Method 1: Mathematical Operation

```
series_num1 - series_num2
```

```
a   -12.0
b     NaN
c     NaN
d     NaN
e   -53.0
f     NaN
g     NaN
i     NaN
o     NaN
u     NaN
dtype: float64
```

Method 2: Calling relevant method

```
series_num1.sub(series_num2, fill_value=0)
```

```
a   -12.0
b   -22.0
c   -33.0
d   -44.0
e   -53.0
f   -66.0
g   -77.0
```

```
i     0.0
o     4.0
u    -3.0
dtype: float64
```

Multiplication of two Series

As mentioned previously, it can be done in *two* different ways, as shown in the following examples:

Method 1: Mathematical operation

```
series_num1 * series_num2
```

```
a    -11.0
b      NaN
c      NaN
d      NaN
e    110.0
f      NaN
g      NaN
i      NaN
o      NaN
u      NaN
dtype: float64
```

Method 2: Calling relevent methode

```
series_num1.mul(series_num2, fill_value=1)
```

```
a    -11.0
b     22.0
c     33.0
d     44.0
e    110.0
f     66.0
g     77.0
i      0.0
o      4.0
u     -3.0
dtype: float64
```

Division of two Series

As mentioned previously, it can be done in *two* different ways, as shown in the following examples:

Method 1: Mathematical operations

```
series_num1 / series_num2
```

```
a   -0.090909
b         NaN
c         NaN
d         NaN
e    0.036364
f         NaN
g         NaN
i         NaN
o         NaN
u         NaN
dtype: float64
```

Mathod 2: Calling relevent methode

```
series_num1.div(series_num2, fill_value=1)
```

```
a   -0.090909
b    0.045455
```

```
c    0.030303
d    0.022727
e    0.036364
f    0.015152
g    0.012987
i    0.000000
o    4.000000
u   -3.000000
dtype: float64
```

As we see all the mathematical operations can be done in 2 method but by using the 2nd methode we can avoid NaN values.

Exercise-1

a) EngAlph: A Series with 26 elements (alphabets) and default index values:

```
import pandas as pd
import string
EngAlph = pd.Series(list(string.ascii_uppercase))
print("EngAlph Series:")
print(EngAlph)
```

```
EngAlph Series:
0     A
1     B
2     C
3     D
4     E
5     F
6     G
7     H
8     I
9     J
10    K
11    L
12    M
13    N
14    O
15    P
16    Q
17    R
18    S
19    T
20    U
21    V
22    W
23    X
24    Y
25    Z
dtype: object
```

b) Vowels: A Series with 5 elements labeled by 'a', 'e', 'i', 'o', 'u', all set to zero. Check if it is an empty series:

```
Vowels = pd.Series(0, index=['a', 'e', 'i', 'o', 'u'])
is_empty_vowels = Vowels.empty
print("\nVowels Series:")
print(Vowels)
print(f"Is Vowels Series empty? {is_empty_vowels}")
```

```

Vowels Series:
a    0
e    0
i    0
o    0
u    0
dtype: int64
Is Vowels Series empty? False
```

c) Friends: A Series from a dictionary with roll numbers as data and first names as keys:

```
friends_dict = {
    'John': 101,
    'Alice': 102,
    'Bob': 103,
    'Cathy': 104,
    'David': 105
}
Friends = pd.Series(friends_dict)
print("\nFriends Series:")
Friends
```

```
Friends Series:
John     101
Alice    102
Bob      103
Cathy    104
David    105
dtype: int64
```

d) MTseries: An empty Series. Check if it is an empty series:

```
MTseries = pd.Series(dtype='float64')
is_empty_mtseries = MTseries.empty
print("\nMTseries:")
print(MTseries)
print(f"Is MTseries empty? {is_empty_mtseries}")
```

```
MTseries:
Series([], dtype: float64)
Is MTseries empty? True
```

e) MonthDays: A Series from a numpy array with the number of days in the 12 months of a year. The labels should be the month numbers from 1 to 12:

```
import numpy as np
days_in_months = np.array([31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31])
MonthDays = pd.Series(days_in_months, index=np.arange(1, 13))
print("\nMonthDays Series:")
MonthDays
```

```
MonthDays Series:
1     31
2     28
3     31
4     30
5     31
6     30
7     31
8     31
9     30
10    31
11    30
12    31
dtype: int32
```

Exercise-2

a) Set all the values of Vowels to 10 and display the Series:

```
Vowels[:] = 10
print("Vowels Series after setting all values to 10:")
print(Vowels)
```

```
Vowels Series after setting all values to 10:
a    10
e    10
i    10
o    10
u    10
dtype: int64
```

b) Divide all values of Vowels by 2 and display the Series:

```
Vowels = Vowels / 2
print("\nVowels Series after dividing all values by 2:")
print(Vowels)
```

```

Vowels Series after dividing all values by 2:
a    5.0
e    5.0
i    5.0
o    5.0
u    5.0
dtype: float64
```

c) Create another series Vowels1 having 5 elements with index labels a, e, i, o and u having values [2,5,6,3,8] respectively:

```
Vowels1 = pd.Series([2, 5, 6, 3, 8], index=['a', 'e', 'i', 'o', 'u'])
print("\nVowels1 Series:")
print(Vowels1)
```

```

Vowels1 Series:
a    2
e    5
i    6
o    3
u    8
dtype: int64
```

d) Add Vowels and Vowels1 and assign the result to Vowels3:

```
Vowels3 = Vowels + Vowels1
print("\nVowels3 Series (Vowels + Vowels1):")
print(Vowels3)
```

```

Vowels3 Series (Vowels + Vowels1):
a     7.0
e    10.0
i    11.0
o     8.0
u    13.0
dtype: float64
```

e) Subtract, Multiply, and Divide Vowels by Vowels1:

```
Vowels_sub = Vowels - Vowels1
print("\nVowels - Vowels1:")
print(Vowels_sub)


Vowels_mul = Vowels * Vowels1
print("\nVowels * Vowels1:")
print(Vowels_mul)


Vowels_div = Vowels / Vowels1
```

```
print("\nVowels / Vowels1:")
print(Vowels_div)
```

```
Vowels - Vowels1:
a    3.0
e    0.0
i   -1.0
o    2.0
u   -3.0
dtype: float64

Vowels * Vowels1:
a    10.0
e    25.0
i    30.0
o    15.0
u    40.0
dtype: float64

Vowels / Vowels1:
a    2.500000
e    1.000000
i    0.833333
o    1.666667
u    0.625000
dtype: float64
```

f) Alter the labels of Vowels1 to [A, E, I, O, U]:

```
Vowels1.index = ['A', 'E', 'I', 'O', 'U']
print("\nVowels1 Series with altered labels:")
print(Vowels1)
```

```
Vowels1 Series with altered labels:
A    2
E    5
I    6
O    3
U    8
dtype: int64
```

Exercise-3

a) Find the dimensions, size, and values of the Series EngAlph, Vowels, Friends, MTseries, and MonthDays:

```
series_list = [EngAlph, Vowels, Friends, MTseries, MonthDays]

# Loop through each series and display its dimensions, size, and values
for series in series_list:
    print(f"\nSeries: {series.name if series.name else 'Unnamed'}")
    print(f"Dimensions: {series.shape}")
    print(f"Size: {series.size}")
    print(f"Values: {series.values}")
```

```
Series: Unnamed
Dimensions: (26,)
Size: 26
Values: ['A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O' 'P' 'Q' 'R'
 'S' 'T' 'U' 'V' 'W' 'X' 'Y' 'Z']

Series: Unnamed
Dimensions: (5,)
Size: 5
Values: [5. 5. 5. 5. 5.]

Series: Unnamed
Dimensions: (5,)
Size: 5
```

```
Values: [101 102 103 104 105]

Series: Unnamed
Dimensions: (0,)
Size: 0
Values: []

Series: Unnamed
Dimensions: (12,)
Size: 12
Values: [31 28 31 30 31 30 31 31 30 31 30 31]
```

b) Rename the Series MTseries as SeriesEmpty:

```
MTseries.name = 'SeriesEmpty'
print(f"\nRenamed Series: {MTseries.name}")
```

```
Renamed Series: SeriesEmpty
```

c) Name the index of the Series MonthDays as monthno and that of Series Friends as Fname:

```
MonthDays.index.name = 'monthno'
print("\nMonthDays Series with index named as 'monthno':")
print(MonthDays)
Friends.index.name = 'Fname'
print("\nFriends Series with index named as 'Fname':")
print(Friends)
```

```
MonthDays Series with index named as 'monthno':
monthno
1     31
2     28
3     31
4     30
5     31
6     30
7     31
8     31
9     30
10    31
11    30
12    31
dtype: int32

Friends Series with index named as 'Fname':
Fname
John     101
Alice    102
Bob      103
Cathy    104
David    105
dtype: int64
```

d) Display the 3rd and 2nd value of the Series Friends, in that order:

```
third_value = Friends.iloc[2]
second_value = Friends.iloc[1]
print("\n3rd value in Friends Series:", third_value)
print("2nd value in Friends Series:", second_value)
```

```
3rd value in Friends Series: 103
2nd value in Friends Series: 102
```

e) Display the alphabets �e� to �p� from the Series EngAlph:

```
alphabets_e_to_p = EngAlph[EngAlph.isin(['E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P'])]
print("\nAlphabets 'e' to 'p' from EngAlph Series:")
print(alphabets_e_to_p)
```

```
Alphabets 'e' to 'p' from EngAlph Series:
4     E
5     F
6     G
7     H
8     I
9     J
10    K
11    L
12    M
13    N
14    O
15    P
dtype: object
```

f) Display the first 10 values in the Series EngAlph:

```
first_10_values = EngAlph.head(10)
print("\nFirst 10 values in EngAlph Series:")
print(first_10_values)
```

```
First 10 values in EngAlph Series:
0    A
1    B
2    C
3    D
4    E
5    F
6    G
7    H
8    I
9    J
dtype: object
```

g) Display the last 10 values in the Series EngAlph:

```
last_10_values = EngAlph.tail(10)
print("\nLast 10 values in EngAlph Series:")
print(last_10_values)
```

```
Last 10 values in EngAlph Series:
16    Q
17    R
18    S
19    T
20    U
21    V
22    W
23    X
24    Y
25    Z
dtype: object
```

h) Display the MTseries:

```
print("\nMTseries (or SeriesEmpty) contents:")
print(MTseries)
```

```
MTseries (or SeriesEmpty) contents:
Series([], Name: SeriesEmpty, dtype: float64)
```

Exercise-4

Create the DataFrame Sales containing year-wise sales figures:

```
import pandas as pd

# Data for the DataFrame
data = {
    '2014': [100.5, 150.8, 200.9, 30000, 40000],
    '2015': [12000, 18000, 22000, 30000, 45000],
    '2016': [20000, 50000, 70000, 100000, 125000],
    '2017': [50000, 60000, 70000, 80000, 90000]
}

# Salesperson names as row labels
index_labels = ['Madhu', 'Kusum', 'Kinshuk', 'Ankit', 'Shruti']

# Create the DataFrame
Sales = pd.DataFrame(data, index=index_labels)
print("Sales DataFrame:")
Sales
```

```
Sales DataFrame:
```

	2014	2015	2016	2017
Madhu	100.5	12000	20000	50000
Kusum	150.8	18000	50000	60000
Kinshuk	200.9	22000	70000	70000
Ankit	30000.0	30000	100000	80000
Shruti	40000.0	45000	125000	90000

Exercise: 5 Perform operations on the Sales DataFrame:

a) Display the row labels of Sales:

```
row_labels = Sales.index
print("\nRow labels of Sales:")
print(row_labels)
```

```

Row labels of Sales:
Index(['Madhu', 'Kusum', 'Kinshuk', 'Ankit', 'Shruti'], dtype='object')
```

b) Display the column labels of Sales:

```
column_labels = Sales.columns
print("\nColumn labels of Sales:")
print(column_labels)
```

```

Column labels of Sales:
Index(['2014', '2015', '2016', '2017'], dtype='object')
```

c) Display the data types of each column of Sales:

```
data_types = Sales.dtypes
print("\nData types of each column in Sales:")
```

```
print(data_types)
```

```
Data types of each column in Sales:
2014    float64
2015      int64
2016      int64
2017      int64
dtype: object
```

d) Display the dimensions, shape, size, and values of Sales:

```
dimensions = Sales.ndim
shape = Sales.shape
size = Sales.size
values = Sales.values

print(f"\nDimensions of Sales: {dimensions}")
print(f"Shape of Sales: {shape}")
print(f"Size of Sales: {size}")
print("Values of Sales:")
print(values)
```

```
Dimensions of Sales: 2
Shape of Sales: (5, 4)
Size of Sales: 20
Values of Sales:
[[1.005e+02 1.200e+04 2.000e+04 5.000e+04]
 [1.508e+02 1.800e+04 5.000e+04 6.000e+04]
 [2.009e+02 2.200e+04 7.000e+04 7.000e+04]
 [3.000e+04 3.000e+04 1.000e+05 8.000e+04]
 [4.000e+04 4.500e+04 1.250e+05 9.000e+04]]
```

e) Display the last two rows of Sales:

```
last_two_rows = Sales.tail(2)
print("\nLast two rows of Sales:")
print(last_two_rows)
```

```
Last two rows of Sales:
           2014   2015    2016   2017
Ankit   30000.0  30000  100000  80000
Shruti  40000.0  45000  125000  90000
```

f) Display the first two columns of Sales:

```
first_two_columns = Sales.iloc[:, :2]
print("\nFirst two columns of Sales:")
print(first_two_columns)
```

```
First two columns of Sales:
            2014   2015
Madhu      100.5  12000
Kusum      150.8  18000
Kinshuk    200.9  22000
Ankit    30000.0  30000
Shruti   40000.0  45000
```

g) Create a dictionary and use it to create a DataFrame Sales2:

```
data_2018 = {
    'Madhu': 160000,
```

```
    'Kusum': 110000,
    'Kinshuk': 500000,
    'Ankit': 340000,
    'Shruti': 900000
}


Sales2 = pd.DataFrame(list(data_2018.values()), index=data_2018.keys(), columns=['2018'])
print("\nSales2 DataFrame:")
print(Sales2)
```

```
Sales2 DataFrame:
           2018
Madhu    160000
Kusum    110000
Kinshuk  500000
Ankit    340000
Shruti   900000
```

h) Check if Sales2 is empty or contains data:

```
is_sales2_empty = Sales2.empty
print(f"\nIs Sales2 empty? {is_sales2_empty}")
```

```
Is Sales2 empty? False
```

Exercise-6 : Use the DataFrame created in Question 5 above todo the following

a) Append the DataFrame Sales2 to the DataFrame Sales:

```
Sales_combined =pd.concat([Sales,Sales2])
print("\nSales DataFrame after appending Sales2:")
Sales_combined
```

```
Sales DataFrame after appending Sales2:
```

	2014	2015	2016	2017	2018
Madhu	100.5	12000.0	20000.0	50000.0	NaN
Kusum	150.8	18000.0	50000.0	60000.0	NaN
Kinshuk	200.9	22000.0	70000.0	70000.0	NaN
Ankit	30000.0	30000.0	100000.0	80000.0	NaN
Shruti	40000.0	45000.0	125000.0	90000.0	NaN
Madhu	NaN	NaN	NaN	NaN	160000.0
Kusum	NaN	NaN	NaN	NaN	110000.0
Kinshuk	NaN	NaN	NaN	NaN	500000.0
Ankit	NaN	NaN	NaN	NaN	340000.0
Shruti	NaN	NaN	NaN	NaN	900000.0

b) Change the DataFrame Sales such that it becomes its transpose:

```
Sales_transposed = Sales.T
print("\nTransposed Sales DataFrame:")
Sales_transposed
```

Transposed Sales DataFrame:

	Madhu	Kusum	Kinshuk	Ankit	Shruti
2014	100.5	150.8	200.9	30000.0	40000.0
2015	12000.0	18000.0	22000.0	30000.0	45000.0
2016	20000.0	50000.0	70000.0	100000.0	125000.0
2017	50000.0	60000.0	70000.0	80000.0	90000.0

c) Display the sales made by all sales persons in the year 2017:

```
sales_2017 = Sales['2017']
print("\nSales made by all sales persons in the year 2017:")
sales_2017
```

```
Sales made by all sales persons in the year 2017:
Madhu      50000
Kusum      60000
Kinshuk    70000
Ankit      80000
Shruti     90000
Name: 2017, dtype: int64
```

e) Display the sales made by Shruti in 2016:

```
sales_shruti_2016 = Sales.loc['Shruti', '2016']
print(f"\nSales made by Shruti in 2016: {sales_shruti_2016}")
```

```
Sales made by Shruti in 2016: 125000
```

g) Delete the data for the year 2014 from the DataFrame Sales:

```
Sales = Sales.drop('2014', axis=1)
print("\nSales DataFrame after deleting the year 2014:")
Sales
```

Sales DataFrame after deleting the year 2014:

	2015	2016	2017
Madhu	12000	20000	50000
Kusum	18000	50000	60000
Kinshuk	22000	70000	70000
Ankit	30000	100000	80000
Shruti	45000	125000	90000

h) Delete the data for salesperson Kinshuk from the DataFrame Sales:

```
Sales = Sales.drop('Kinshuk', axis=0)
print("\nSales DataFrame after deleting Kinshuk's data:")
Sales
```

```
Sales DataFrame after deleting Kinshuk's data:
```

	2015	2016	2017
Madhu	12000	20000	50000
Kusum	18000	50000	60000
Ankit	30000	100000	80000
Shruti	45000	125000	90000

i) Change the name of the salesperson Ankit to Vivaan and Madhu to Shailesh

```
Sales = Sales.rename(index={'Ankit': 'Vivaan', 'Madhu': 'Shailesh'})
print("\nSales DataFrame after renaming salespersons:")
Sales
```

```
Sales DataFrame after renaming salespersons:
```

	2015	2016	2017
Shailesh	12000	20000	50000
Kusum	18000	50000	60000
Vivaan	30000	100000	80000
Shruti	45000	125000	90000

j) Update the sale made by Shailesh in 2018 to 100000:

```
Sales.loc['Shailesh', '2018'] = 100000
print("\nSales DataFrame after updating Shailesh's 2018 sales:")
Sales
```

```
Sales DataFrame after updating Shailesh's 2018 sales:
```

	2015	2016	2017	2018
Shailesh	12000	20000	50000	100000.0
Kusum	18000	50000	60000	NaN
Vivaan	30000	100000	80000	NaN
Shruti	45000	125000	90000	NaN

k) Write the values of DataFrame Sales to a comma-separated file SalesFigures.csv on the disk without row labels and column labels:

```
# Write Sales DataFrame to a CSV file without row and column labels
Sales.to_csv('SalesFigures.csv', header=False, index=False)
print("\nSales DataFrame written to SalesFigures.csv")
```

```
Sales DataFrame written to SalesFigures.csv
```

l) Read the data from SalesFigures.csv into a DataFrame SalesRetrieved and display it. Update row labels and column labels of SalesRetrieved to match Sales:

```
SalesRetrieved = pd.read_csv('SalesFigures.csv', header=None)


SalesRetrieved.index = Sales.index
SalesRetrieved.columns = Sales.columns

print("\nSalesRetrieved DataFrame:")
```

```
SalesRetrieved
```

```
SalesRetrieved DataFrame:
```

	2015	2016	2017	2018
Shailesh	12000	20000	50000	100000.0
Kusum	18000	50000	60000	NaN
Vivaan	30000	100000	80000	NaN
Shruti	45000	125000	90000	NaN

END OF SERIES Assignment

Double-click (or enter) to edit

END OF Assignment