Pandas contain data structures and data manipulation tools designed for data cleaning and analysis.

While pandas adopt much code from NumPy, the difference is that Pandas is designed for tabular, heterogeneous data. NumPy, by difference, is best suited for working with homogeneous numerical array data.

The name Pandas is derived from the term 'panel data' (an econometrics term for multidimensional structured data sets).

mport the pandas library; the following convention is used

In [1]:
```
import pandas as pd
```

Data Structures

Pandas has two data structures as follows:

1. A Series is a 1-dimensional labelled array that can hold data of any type (such as integer, string, boolean, float, python objects). Its axis labels are collectively called an index.
2. A DataFrame is a 2-dimensional labelled data structure with columns. It supports multiple data types.

Pandas Series

Pandas Series is a one-dimensional labelled array capable of holding any data type.
However, a series is a sequence of similar data types, similar to an array, list, or column in a table.
It will assign a labelled index to each item in the pd.Series. By default, each item will receive an index label from 0 to N, where N is the length of the Series minus one.

Creating a Series

To create a numeric series

In [3]:
```
# create a numeric series
numbers = range(1,10)
# covert it to series
pd.Series(numbers)
```

Out[3]:
```
0    1
1    2
2    3
3    4
4    5
5    6
6    7
7    8
8    9
dtype: int64
```

To create an object series

In [4]:
```
# create an object series
string = "Hi" , "How" ,"are", "you", "?"


# create a series from the above list
pd.Series(string)
```

Out[4]:
```
0     Hi
1    How
2    are
3    you
4      ?
dtype: object
```

To create a series by giving both numeric and string values

In [5]:
```
# create a Series with an arbitrary list
pd.Series([345, 'London', 34.5, -34.45, 'Happy Birthday'])
```

Out[5]:
```
0              345
1           London
2             34.5
3           -34.45
4    Happy Birthday
dtype: object
```

Here the numeric values are treated as object.

To set index values for a series

In [6]:
```
# declare a list of marks
marks = [60, 89, 74, 86]

# declare a list of subjects
subject = ["Maths", "Science", "English" , "Social Science"]

# create a series from the above list of marks with subjects as its index names
# index: adds the index
marks_series = pd.Series(marks, index = subject)
marks_series
```

Out[6]:
```
Maths             60
Science           89
English           74
Social Science    86
dtype: int64
```

The index is added using the argument index=. The data type of the series continues to be numeric.

To print the values and index of the Series

In [7]:
```
# print the index of the series
marks_series.index
```

Out[7]:
```
Index(['Maths', 'Science', 'English', 'Social Science'], dtype='object')
```

In [8]:
```
# prints the values of the series
marks_series.values
```

Out[8]:
```
array([60, 89, 74, 86], dtype=int64)
```

To create a series from a dictionary

In [9]:
```
# declare a dictionary
data = {'Maths': 60, 'Science': 89, 'English': 76, 'Social Science': 86}

# create a series from the dictionary
# the dictionary keys are the index names
# the dictionary values are the series values
pd.Series(data)
```

Out[9]:
```
Maths             60
Science           89
English           76
Social Science    86
dtype: int64
```

To create a series using library numpy

In [10]:
```
# import the numpy library
import numpy as np

# declare the variable 'sequence' using linspace()
sequence = np.linspace(0,10, 5)

# print the array 'sequence'
sequence
```

Out[10]:
```
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

In [11]:
```
# create a series from the above sequence
pd.Series(sequence)
```

Out[11]:
```
0     0.0
1     2.5
2     5.0
3     7.5
4    10.0
dtype: float64
```

A series with missing values

If we pass a key that is not defined, then its value will be NaN.

In [12]:
```
# create a list of subjects
subjects = ["Maths", "Science", "Art and Craft" , "Social Science"]

# declare a dictionary
data = {'Maths': 60, 'Science': 89, 'English': 76, 'Social Science': 86}

# create a series from the dictionary
marks_series = pd.Series(data, index = subjects)

# print series
print(marks_series)
```

```
Maths             60.0
Science           89.0
Art and Craft      NaN
Social Science    86.0
dtype: float64
```

In [ ]:
```

```

Manipulating Series

To check for null values using `.isnull`

In [13]:
```
# check for nulls in the data
marks_series.isnull()
```

Out[13]:
```
Maths             False
Science           False
Art and Craft      True
Social Science    False
dtype: bool
```

`False` indicates that the value is not null.

To check for null values using `.notnull`

In [16]:
```
# check for nulls in the data
marks_series.notnull()
```

Out[16]:
```
Maths              True
Science            True
Art and Craft     False
Social Science     True
dtype: bool
```

`True` indicates that the value is not null.

To know the subjects in which marks score is more than 75

In [17]:
```
# check for marks more than 75
marks_series[marks_series > 75]
```

Out[17]:
```
Science           89.0
Social Science    86.0
dtype: float64
```

To assign 68 marks to 'Art and Craft'

In [18]:
```
# assign 68 marks to 'Art and Craft'
marks_series["Art and Craft"] = 68

# print the series
marks_series
```

Out[18]:
```
Maths             60.0
Science           89.0
Art and Craft     68.0
Social Science    86.0
dtype: float64
```

To check whether Maths marks are 73

In [20]:
```
# check whether Maths marks are 73
marks_series.Maths == 73
```

Out[20]:
```
False
```

In [19]:
```
# or you may use
marks_series["Maths"] == 73
```

Out[19]:
```
False
```

To create a series by generating numpy random numbers

In [31]:
```
# create the numbers series
# generate a sequence of 15 random numbers using random()
# round(): rounds off the number to nearest integer
num = pd.Series(np.random.random(15)*10).round()
```

To find the square of the numbers series

In [32]:
```
# declare a variable square
# variable 'square' contains the
square = pd.Series(num*num)
square.index = [num]
square
```

Out[32]:
```
7.0    49.0
4.0    16.0
9.0    81.0
6.0    36.0
7.0    49.0
2.0     4.0
5.0    25.0
4.0    16.0
4.0    16.0
1.0     1.0
7.0    49.0
7.0    49.0
6.0    36.0
4.0    16.0
3.0     9.0
dtype: float64
```

To assign index name and object name

In [33]:
```
# assign object name
square.name = 'Square'
```

```
# assign index name
square.index.name = 'Number'

# print the series
square
```

Out[33]:
```
7.0    49.0
4.0    16.0
9.0    81.0
6.0    36.0
7.0    49.0
2.0     4.0
5.0    25.0
4.0    16.0
4.0    16.0
1.0     1.0
7.0    49.0
7.0    49.0
6.0    36.0
4.0    16.0
3.0     9.0
Name: Square, dtype: float64
```

Note: A Series's index can be altered in-place by assignment

From the output, it is not clear that the index column is labeled, to check whether it is labeled let us print it

We use `Series.index` to print the index.

In [34]:
```
# print the index
square.index
```

Out[34]:
```
MultiIndex([(7.0,),
            (4.0,),
            (9.0,),
            (6.0,),
            (7.0,),
            (2.0,),
            (5.0,),
            (4.0,),
            (4.0,),
            (1.0,),
            (7.0,),
            (7.0,),
            (6.0,),
            (4.0,),
            (3.0,)],
           name='Number')
```

From `name='Number'`, it is seen that the column is labeled.

Add a number 5 to every element of the series

In [35]:
```
# add 5 to each element
square + 5
```

Out[35]:
```
7.0    54.0
4.0    21.0
9.0    86.0
6.0    41.0
7.0    54.0
2.0     9.0
5.0    30.0
4.0    21.0
4.0    21.0
1.0     6.0
7.0    54.0
7.0    54.0
6.0    41.0
4.0    21.0
3.0    14.0
Name: Square, dtype: float64
```

To extract a value specifying the index

In [36]:
```
# obtain the value having index 7
square[7]
```

Out[36]:
```
7.0    49.0
7.0    49.0
7.0    49.0
7.0    49.0
Name: Square, dtype: float64
```

All values with index as '7' are obtained

To extract a range of values specifying the location

In [39]:
```
# extract values having the location
# obtain the values from the 3rd position till the 6th position
square[3:7]
```

Out[39]:
```
6.0    36.0
7.0    49.0
2.0     4.0
5.0    25.0
Name: Square, dtype: float64
```

Usage of `.iloc`

We use `.iloc` to get the values of the specified index of numbers

In [37]:
```
# obtain the value in the 5th position
square.iloc[5]
```

Out[37]:
```
4.0
```

In [38]:
```
# obtain the values from the 3rd till the 8th position
square.iloc[3:9]
```

Out[38]:
```
6.0    36.0
7.0    49.0
2.0     4.0
5.0    25.0
4.0    16.0
4.0    16.0
Name: Square, dtype: float64
```

Sorting a numeric series

In [40]:
```
# create a pandas series
age = pd.Series([23, 45, np.nan, 41, 23, 34, 55, np.nan, 34, 20])

# print values
age
```

Out[40]:
```
0    23.0
1    45.0
2     NaN
3    41.0
4    23.0
5    34.0
6    55.0
7     NaN
8    34.0
9    20.0
dtype: float64
```

In [41]:
```
# ascending order
# sort_values: sorts the values
# ascending : if specified as True, it sorts values in ascending order (default val
age.sort_values(ascending = True)
```

Out[41]:
```
9    20.0
0    23.0
4    23.0
5    34.0
8    34.0
3    41.0
1    45.0
6    55.0
2     NaN
7     NaN
dtype: float64
```

In [42]:
```
# arrange in descending order
# sort_values: sorts the values
# ascending : if specified as True, it sorts values in ascending order (default val
# set ascending to False to sort the values in ascending order
age.sort_values(ascending = False)
```

Out[42]:
```
6    55.0
1    45.0
3    41.0
5    34.0
8    34.0
0    23.0
4    23.0
9    20.0
2     NaN
7     NaN
dtype: float64
```

Sorting a categorical series

In [43]:
```
# create a pandas series
string_values = pd.Series(["a", "j", "d", "f", "t", "a"])

# print the values
string_values
```

Out[43]:
```
0    a
1    j
2    d
3    f
4    t
5    a
dtype: object
```

In [44]:
```
# ascending order
# sort_values: sorts the values
# ascending : if specified as True, it sorts values in ascending order (default val
string_values.sort_values(ascending = True)
```

Out[44]:
```
0    a
5    a
2    d
3    f
1    j
4    t
dtype: object
```

In [45]:
```
# descending order
# sort_values: sorts the values
# ascending : if specified as True, it sorts values in ascending order (default val
# set ascending to False to sort the values in ascending order
string_values.sort_values(ascending = False)
```

Out[45]:
```
4    t
1    j
3    f
2    d
0    a
5    a
dtype: object
```

Sorting based on index

In [46]:
```
# recall the series 'square'
square
```

Out[46]:
```
7.0    49.0
4.0    16.0
9.0    81.0
6.0    36.0
7.0    49.0
2.0     4.0
5.0    25.0
4.0    16.0
4.0    16.0
1.0     1.0
7.0    49.0
7.0    49.0
6.0    36.0
4.0    16.0
3.0     9.0
Name: Square, dtype: float64
```

In [47]:
```
# sort in ascending order based on index
# sort_index: sorts the series based on the index
# ascending : if specified as True, it sorts values in ascending order (default val
square.sort_index(ascending = True)
```

Out[47]:
```
1.0     1.0
2.0     4.0
3.0     9.0
4.0    16.0
4.0    16.0
4.0    16.0
4.0    16.0
5.0    25.0
6.0    36.0
6.0    36.0
7.0    49.0
7.0    49.0
7.0    49.0
7.0    49.0
9.0    81.0
Name: Square, dtype: float64
```

In [48]:
```
# sort in descending order based on index
# sort_index: sorts the series based on the index
# ascending : if specified as True, it sorts values in ascending order (default val
# set ascending to False to sort the values in ascending order
square.sort_index(ascending = False)
```

Out[48]:
```
9.0    81.0
7.0    49.0
7.0    49.0
7.0    49.0
7.0    49.0
6.0    36.0
6.0    36.0
5.0    25.0
4.0    16.0
4.0    16.0
4.0    16.0
4.0    16.0
3.0     9.0
2.0     4.0
1.0     1.0
Name: Square, dtype: float64
```

Rank a Series

In [50]:
```
# recall the marks_series
marks_series
```

Out[50]:
```
Maths            60.0
Science          89.0
Art and Craft    68.0
Social Science   86.0
dtype: float64
```

In [51]:
```
# rank the marks in ascending order
# rank(): ranks the values of a series
marks_series.rank()
```

Out[51]:
```
Maths            1.0
Science          4.0
Art and Craft    2.0
Social Science   3.0
dtype: float64
```

In [52]:
```
# rank the marks in ascending order
# rank(): ranks the values of a series
# ascending : if specified as True, it sorts values in ascending order (default val
```

```
# set ascending to False to sort the values in ascending order
marks_series.rank(ascending = False)
```

Out[52]:
```
Maths            4.0
Science          1.0
Art and Craft    3.0
Social Science   2.0
dtype: float64
```

In [ ]:
```

```

4. Pandas DataFrames

A DataFrame is a tabular representation of data containing an ordered collection of columns, each of which can be a different type (such as numeric, string, boolean).

The DataFrame has both row and column index; it can be thought of as a dict of Series all sharing the same index. In a data frame, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays.

While a DataFrame is physically two-dimensional, it can be used to represent higher dimensional data in a tabular format using hierarchical indexing.

4.1 Creating DataFrames

Creating a data frame a dictionary

In [54]:
```
# create a dictionary
data = {'Subject': ['Maths', 'History', 'Science', 'English', 'Georaphy', 'Art'],
        'Marks': (45, 65, 78, 65, 80, 78),
        'GPA': [2.5, 3.0, 3.5, 2.0, 4.0, 4.0]}

# create the dataframe using DataFrame()
df_marks = pd.DataFrame(data)

# print the dataframe
print(df_marks)
```

```
    Subject  Marks  GPA
0     Maths     45  2.5
1   History     65  3.0
2   Science     78  3.5
3   English     65  2.0
4  Georaphy     80  4.0
5       Art     78  4.0
```

Note: Like Series, the resulting DataFrame is assigned index automatically. And the 'Marks' values are in a tuple.

Another way to create dataframe from dictionary

In [ ]:
```
# creare the dictionary
data = [{'Subject': 'Maths', 'Marks': 45, 'GPA':2.5},
        {'Subject':'History', 'Marks':65, 'GPA':3.0},
        {'Subject':'Science', 'Marks':78, 'GPA':3.5},
        {'Subject':'English', 'Marks':65, 'GPA':2.0},
        {'Subject':'Georaphy', 'Marks':80, 'GPA':4.0},
```

```
        {'Subject':'Art', 'Marks':78, 'GPA':4.0}]

# create the dataframe using DataFrame()
df_marks = pd.DataFrame(data)

# print the dataframe
print(df_marks)
```

To create dataframe from lists

In [56]:
```
# declare the list 'Subject'
Subject = ['Maths', 'History', 'Science', 'English', 'Georaphy', 'Art']

# declare the list 'Marks'
Marks = [45, 65, 78, 65, 80, 78]

# declare the list 'CPA'
GPA = [2.5, 3.0, 3.5, 2.0, 4.0, 4.0]
```

In [57]:
```
# create a DataFrame from the lists
# index: specifies the index names
df_marks = pd.DataFrame([Subject, Marks, GPA], index = ['Subject','Marks','GPA'])

# print the DataFrame
df_marks
```

Out[57]:

	0	1	2	3	4	5
Subject	Maths	History	Science	English	Georaphy	Art
Marks	45	65	78	65	80	78
GPA	2.5	3.0	3.5	2.0	4.0	4.0

However to want a vertical dataframe so we use `.T`. The 'T' stands for transpose.

In [58]:
```
# transpose the DataFrame
df_marks.T
```

Out[58]:

	Subject	Marks	GPA
0	Maths	45	2.5
1	History	65	3.0
2	Science	78	3.5
3	English	65	2.0
4	Georaphy	80	4.0
5	Art	78	4.0

To create dataframe from series

In [59]:
```
# declare the series 'Subject'
Subject = pd.Series(['Maths', 'History', 'Science', 'English', 'Georaphy', 'Art'])

# declare the series 'Marks'
Marks = pd.Series([45, 65, 78, 65, 80, 78])
```

```
# declare the series 'GPA'
GPA = pd.Series([2.5, 3.0, 3.5, 2.0, 4.0, 4.0])
```

In [60]:
```
# create a DataFrame from the Series
# index: specifies the index names
df_marks = pd.DataFrame([Subject, Marks, GPA], index = ['Subject','Marks','GPA'])

# print the DataFrame
df_marks
```

Out[60]:

	0	1	2	3	4	5
Subject	Maths	History	Science	English	Georaphy	Art
Marks	45	65	78	65	80	78
GPA	2.5	3.0	3.5	2.0	4.0	4.0

However to want a vertical dataframe so we use `.T`. The 'T' stands for transpose.

In [61]:
```
# transpose the DataFrame
df_marks.T
```

Out[61]:

	Subject	Marks	GPA
0	Maths	45	2.5
1	History	65	3.0
2	Science	78	3.5
3	English	65	2.0
4	Georaphy	80	4.0
5	Art	78	4.0

Remark: Assign a name to the data frame and then use `.T` to transpose it.

To read data from csv file

In [63]:
```
df_basic_info = pd.read_csv("basic_info.csv")
```

In [64]:
```
type(df_basic_info)
```

Out[64]:
```
pandas.core.frame.DataFrame
```

On checking the data type, we notice it is read as pandas data frame.

In [65]:
```
print(df_basic_info)
```

```
    Age  Weight (in kg)  Height (in m)
0    45              60          1.35
1    12              43          1.21
2    54              78          1.50
3    26              65          1.21
4    68              50          1.32
5    21              43          1.52
6    10              32          1.65
7    57              34          1.61
8    75              23          1.24
9    32              21          1.52
10   23              53          1.50
11   34              65          1.76
12   55              89          1.65
13   23              45          1.75
14   56              76          1.69
15   67              78          1.85
16   26              65          1.21
17   56              74          1.69
18   67              78          1.85
19   26              65          1.21
20   68              50          1.32
21   56              76          1.69
22   67              78          1.85
```

To print head of the data

In [66]:
```
# display the first 5 rows of the DataFrame
# head(): displays the first 5 rows
# to display more rows, for example 15, use head(15)
# the default value is 5
df_basic_info.head()
```

Out[66]:

	Age	Weight (in kg)	Height (in m)
0	45	60	1.35
1	12	43	1.21
2	54	78	1.50
3	26	65	1.21
4	68	50	1.32

By default, the `.head()` will display **first** five rows. However, we can set the desired number of rows to be displayed.

Say we want to see the first 9 rows, we write the number 9 in the parentheses.

In [68]:
```
# display 9 rows
df_basic_info.head(9)
```

Out[68]:

	Age	Weight (in kg)	Height (in m)
0	45	60	1.35
1	12	43	1.21
2	54	78	1.50
3	26	65	1.21
4	68	50	1.32
5	21	43	1.52
6	10	32	1.65
7	57	34	1.61
8	75	23	1.24

To print tail of the data

In [69]:
```
# to display the last 5 rows
df_basic_info.tail()
```

Out[69]:

	Age	Weight (in kg)	Height (in m)
18	67	78	1.85
19	26	65	1.21
20	68	50	1.32
21	56	76	1.69
22	67	78	1.85

By default, the `.tail()` will display **last** five rows. However, we can set the desired number of rows to be displayed.

Say we want to see the last 14 rows, we write the number 14 in the parentheses.

In [70]:
```
# to display the last 14 rows
df_basic_info.tail(14)
```

Out[70]:

	Age	Weight (in kg)	Height (in m)
9	32	21	1.52
10	23	53	1.50
11	34	65	1.76
12	55	89	1.65
13	23	45	1.75
14	56	76	1.69
15	67	78	1.85
16	26	65	1.21
17	56	74	1.69
18	67	78	1.85
19	26	65	1.21
20	68	50	1.32
21	56	76	1.69
22	67	78	1.85

To obtain the dimension of the data

In [71]:
```
# to display the shape of the data
df_basic_info.shape
```

Out[71]:
```
(23, 3)
```

To know the data types of a data frame

In [73]:
```
# to know the type of each variable
df_basic_info.dtypes
```

Out[73]:
```
Age                 int64
Weight (in kg)      int64
Height (in m)     float64
dtype: object
```

We see the data type of each variable.

To know some information of the data

In [74]:
```
# to know information on the variables in the data
df_basic_info.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23 entries, 0 to 22
Data columns (total 3 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Age             23 non-null     int64  
 1   Weight (in kg)  23 non-null     int64  
 2   Height (in m)   23 non-null     float64
dtypes: float64(1), int64(2)
memory usage: 680.0 bytes
```

We see this output gives the number of rows present in the data `RangeIndex: 23 entries, 0 to 22` There are 23 rows numbered from 0 to 22. And there are a total of three columns - `Data columns (total 3 columns)` .

Consider `Age 23 non-null int64` indicates that the column named 'Age' has 23 non-null observations having the data type 'int64'

And finally the memory used to save this dataframe is 680 bytes.

To check the data type of column in the data frame

In [75]:
```
# check the type of each variable
type(df_basic_info.Age)
```

Out[75]:
```
pandas.core.series.Series
```

In [76]:
```
# check the type of each variable
type(df_basic_info["Weight (in kg)"])
```

Out[76]:
```
pandas.core.series.Series
```

In [77]:
```
# check the type of each variable
type(df_basic_info["Height (in m)"])
```

Out[77]:
```
pandas.core.series.Series
```

Note that every column of the data frame is a pandas Series.

Manipulating DataFrames

Add new column and rows

Remark:

1. DataFrame[column] works for any column name, but DataFrame.column only works when the column name is a valid Python variable name.
2. New columns cannot be created with the `df_basic_info.BMI` syntax.

Adding a new column to the data set

In [80]:
```
# create a new variable BMI
df_basic_info["BMI"] = df_basic_info["Weight (in kg)"] / df_basic_info["Height (in

# print the DataFrame
df_basic_info.head()
```

Out[80]:

	Age	Weight (in kg)	Height (in m)	BMI
0	45	60	1.35	32.921811
1	12	43	1.21	29.369579
2	54	78	1.50	34.666667
3	26	65	1.21	44.395875
4	68	50	1.32	28.696051

In [81]:
```
# check the shape of the DataFrame
df_basic_info.shape
```

Out[81]:
```
(23, 4)
```

Adding a new row to the data set

A new row can be added using the `loc`

In [82]:
```
# add a new row at the end of the DataFrame
df_basic_info.loc[24] = [45, 85, 1.8, 26.3]

# display the DataFrame
df_basic_info
```

Out[82]:

	Age	Weight (in kg)	Height (in m)	BMI
0	45.0	60.0	1.35	32.921811
1	12.0	43.0	1.21	29.369579
2	54.0	78.0	1.50	34.666667
3	26.0	65.0	1.21	44.395875
4	68.0	50.0	1.32	28.696051
5	21.0	43.0	1.52	18.611496
6	10.0	32.0	1.65	11.753903
7	57.0	34.0	1.61	13.116778
8	75.0	23.0	1.24	14.958377
9	32.0	21.0	1.52	9.089335
10	23.0	53.0	1.50	23.555556
11	34.0	65.0	1.76	20.983988
12	55.0	89.0	1.65	32.690542
13	23.0	45.0	1.75	14.693878
14	56.0	76.0	1.69	26.609713
15	67.0	78.0	1.85	22.790358
16	26.0	65.0	1.21	44.395875
17	56.0	74.0	1.69	25.909457
18	67.0	78.0	1.85	22.790358
19	26.0	65.0	1.21	44.395875
20	68.0	50.0	1.32	28.696051
21	56.0	76.0	1.69	26.609713
22	67.0	78.0	1.85	22.790358
24	45.0	85.0	1.80	26.300000

We see that a new column number 23 has be added to the data.

Indexing a dataframe using `.iloc`

`DataFrame.iloc[]` method is used when the index label of a data frame is something other than numeric series of 0, 1, 2, 3....n or in case the user doesn't know the index label.

We shall work on the BMI data set.

Select the 2nd row

In [83]:
```
# select the second row
df_basic_info.iloc[2]
```

Out[83]:
```
Age                54.000000
Weight (in kg)     78.000000
Height (in m)       1.500000
BMI                34.666667
Name: 2, dtype: float64
```

Select fourth, seventh and tenth rows

In [103…
```
# select the 4th, 7th and the 10th row
df_basic_info.iloc[[4,7,10]]
```

Out[103]:

	Age	Weight (in kg)	Height (in m)	BMI	Rank_min	Rank_densed
4	68.0	50.0	1.32	28.696051	8.0	13.0
7	57.0	34.0	1.61	13.116778	22.0	3.0
10	23.0	53.0	1.50	23.555556	14.0	9.0

We use two square brackets since we are passing a list of row numbers to be accessed.

Select 12th to 17th rows

In [102…
```
# select rows
df_basic_info.iloc[12:18]
```

Out[102]:

	Age	Weight (in kg)	Height (in m)	BMI	Rank_min	Rank_densed
12	55.0	89.0	1.65	32.690542	6.0	15.0
13	23.0	45.0	1.75	14.693878	21.0	4.0
14	56.0	76.0	1.69	26.609713	10.0	12.0
15	67.0	78.0	1.85	22.790358	15.0	8.0
16	26.0	65.0	1.21	44.395875	1.0	18.0
17	56.0	74.0	1.69	25.909457	13.0	10.0

Select the 1st column

In [101…
```
# select the 1st column
df_basic_info.iloc[:, 1]
```

Out[101]:
```
0     60.0
1     43.0
2     78.0
3     65.0
4     50.0
5     43.0
6     32.0
7     34.0
8     23.0
9     21.0
10    53.0
11    65.0
12    89.0
13    45.0
14    76.0
15    78.0
16    65.0
17    74.0
18    78.0
19    65.0
20    50.0
21    76.0
22    78.0
24    85.0
Name: Weight (in kg), dtype: float64
```

Select the last column

In [104…
```
# select the last column
# the colon indicates that all the rows are selected
# -1 indicated that the last column is selected
df_basic_info.iloc[:,-1]
```

Out[104]:
```
0     16.0
1     14.0
2     17.0
3     18.0
4     13.0
5      6.0
6      2.0
7      3.0
8      5.0
9      1.0
10     9.0
11     7.0
12    15.0
13     4.0
14    12.0
15     8.0
16    18.0
17    10.0
18     8.0
19    18.0
20    13.0
21    12.0
22     8.0
24    11.0
Name: Rank_densed, dtype: float64
```

To select the last column we use -1, to select the second last column we use -2

Select the first two columns

In [105…
```
# select the 1st and 2nd columns
# the colon indicates that all the rows are selected
# 0:2 indicates that the 1st and 2nd columns are selected
df_basic_info.iloc[:,0:2]
```

Out[105]:

	Age	Weight (in kg)
0	45.0	60.0
1	12.0	43.0
2	54.0	78.0
3	26.0	65.0
4	68.0	50.0
5	21.0	43.0
6	10.0	32.0
7	57.0	34.0
8	75.0	23.0
9	32.0	21.0
10	23.0	53.0
11	34.0	65.0
12	55.0	89.0
13	23.0	45.0
14	56.0	76.0
15	67.0	78.0
16	26.0	65.0
17	56.0	74.0
18	67.0	78.0
19	26.0	65.0
20	68.0	50.0
21	56.0	76.0
22	67.0	78.0
24	45.0	85.0

Select the first two columns and 5 to 10 rows

In [106…
```
# 5:11 indicates that the 5th to 10th rows will be selected
# 0:2 indicates that the 1st and 2nd columns be selected
df_basic_info.iloc[5:11, 0:2]
```

Out[106]:

	Age	Weight (in kg)
5	21.0	43.0
6	10.0	32.0
7	57.0	34.0
8	75.0	23.0
9	32.0	21.0
10	23.0	53.0

Indexing a dataframe using `.loc`

`DataFrame.loc[]` method is a method that takes only index labels and returns row or dataframe if the index label exists in the caller data frame.
`DataFrame.loc[Row_names, column_names]` is used to select or index rows or columns based on their name.

Select 1 to 5 rows and 2nd and 4th columns

In [107…

```
# 1:5 indicates that rows with indices 1, 2, 3, 4 and 5 are selected
# ["Weight (in kg)","BMI"] indicates that the specified columns be selected
df_basic_info.loc[1:5,["Weight (in kg)","BMI"]]
```

Out[107]:

	Weight (in kg)	BMI
1	43.0	29.369579
2	78.0	34.666667
3	65.0	44.395875
4	50.0	28.696051
5	43.0	18.611496

Note: the row names are numbers

Selecting columns by specifying column names

Select the column 'Age'

In [108…

```
# select the coumn 'Age'
df_basic_info.Age
```

Out[108]:
```
0     45.0
1     12.0
2     54.0
3     26.0
4     68.0
5     21.0
6     10.0
7     57.0
8     75.0
9     32.0
10    23.0
11    34.0
12    55.0
13    23.0
14    56.0
15    67.0
16    26.0
17    56.0
18    67.0
19    26.0
20    68.0
21    56.0
22    67.0
24    45.0
Name: Age, dtype: float64
```

Remark: Using this method we can select only one column.

In [109…
```
# OR
df_basic_info["Age"]
```

Out[109]:
```
0     45.0
1     12.0
2     54.0
3     26.0
4     68.0
5     21.0
6     10.0
7     57.0
8     75.0
9     32.0
10    23.0
11    34.0
12    55.0
13    23.0
14    56.0
15    67.0
16    26.0
17    56.0
18    67.0
19    26.0
20    68.0
21    56.0
22    67.0
24    45.0
Name: Age, dtype: float64
```

Select the column 'Age' and 'BMI'

In [110…
```
# select two columns
# the column names are passed in a list
df_basic_info[["Age","BMI"]]
```

Out[110]:

	Age	BMI
0	45.0	32.921811
1	12.0	29.369579
2	54.0	34.666667
3	26.0	44.395875
4	68.0	28.696051
5	21.0	18.611496
6	10.0	11.753903
7	57.0	13.116778
8	75.0	14.958377
9	32.0	9.089335
10	23.0	23.555556
11	34.0	20.983988
12	55.0	32.690542
13	23.0	14.693878
14	56.0	26.609713
15	67.0	22.790358
16	26.0	44.395875
17	56.0	25.909457
18	67.0	22.790358
19	26.0	44.395875
20	68.0	28.696051
21	56.0	26.609713
22	67.0	22.790358
24	45.0	26.300000

Conditional subsetting

Selecting rows where the value of `Age` is greater than 47

In [89]:
```
# to selct rows where the Age is greater than 47
df_basic_info[df_basic_info['Age'] > 47]
```

Out[89]:

	Age	Weight (in kg)	Height (in m)	BMI
2	54.0	78.0	1.50	34.666667
4	68.0	50.0	1.32	28.696051
7	57.0	34.0	1.61	13.116778
8	75.0	23.0	1.24	14.958377
12	55.0	89.0	1.65	32.690542
14	56.0	76.0	1.69	26.609713
15	67.0	78.0	1.85	22.790358
17	56.0	74.0	1.69	25.909457
18	67.0	78.0	1.85	22.790358
20	68.0	50.0	1.32	28.696051
21	56.0	76.0	1.69	26.609713
22	67.0	78.0	1.85	22.790358

Subsetting the age more than 40 *or* weight column value more than 65

In [111…

```
# to select rows where either age is greater than 40 or weight is more than 65
df_basic_info[ (df_basic_info["Age"] > 40) | (df_basic_info['Weight (in kg)'] > 65)
```

Out[111]:

	Age	Weight (in kg)	Height (in m)	BMI	Rank_min	Rank_densed
0	45.0	60.0	1.35	32.921811	5.0	16.0
2	54.0	78.0	1.50	34.666667	4.0	17.0
4	68.0	50.0	1.32	28.696051	8.0	13.0
7	57.0	34.0	1.61	13.116778	22.0	3.0
8	75.0	23.0	1.24	14.958377	20.0	5.0
12	55.0	89.0	1.65	32.690542	6.0	15.0
14	56.0	76.0	1.69	26.609713	10.0	12.0
15	67.0	78.0	1.85	22.790358	15.0	8.0
17	56.0	74.0	1.69	25.909457	13.0	10.0
18	67.0	78.0	1.85	22.790358	15.0	8.0
20	68.0	50.0	1.32	28.696051	8.0	13.0
21	56.0	76.0	1.69	26.609713	10.0	12.0
22	67.0	78.0	1.85	22.790358	15.0	8.0
24	45.0	85.0	1.80	26.300000	12.0	11.0

Subsetting the age *and* weight column value more than 65

In [91]:

```
# select rows where both age and weight are more than 65
df_basic_info[(df_basic_info["Age"] > 65) & (df_basic_info['Weight (in kg)'] > 65)]
```

Out[91]:

	Age	Weight (in kg)	Height (in m)	BMI
15	67.0	78.0	1.85	22.790358
18	67.0	78.0	1.85	22.790358
22	67.0	78.0	1.85	22.790358

Sort the data frame on the basis of values in a column

Each column of a pandas DataFrame is treated as a pandas Series. The `.sort_values()` in DataFrames works similar to the `pandas.Series`.

In [92]:
```
# sort the data frame on basis of 'Age' values
# by default the values will get sorted in ascending order
df_basic_info.sort_values('Age')
```

Out[92]:

	Age	Weight (in kg)	Height (in m)	BMI
6	10.0	32.0	1.65	11.753903
1	12.0	43.0	1.21	29.369579
5	21.0	43.0	1.52	18.611496
13	23.0	45.0	1.75	14.693878
10	23.0	53.0	1.50	23.555556
19	26.0	65.0	1.21	44.395875
3	26.0	65.0	1.21	44.395875
16	26.0	65.0	1.21	44.395875
9	32.0	21.0	1.52	9.089335
11	34.0	65.0	1.76	20.983988
0	45.0	60.0	1.35	32.921811
24	45.0	85.0	1.80	26.300000
2	54.0	78.0	1.50	34.666667
12	55.0	89.0	1.65	32.690542
14	56.0	76.0	1.69	26.609713
17	56.0	74.0	1.69	25.909457
21	56.0	76.0	1.69	26.609713
7	57.0	34.0	1.61	13.116778
22	67.0	78.0	1.85	22.790358
15	67.0	78.0	1.85	22.790358
18	67.0	78.0	1.85	22.790358
4	68.0	50.0	1.32	28.696051
20	68.0	50.0	1.32	28.696051
8	75.0	23.0	1.24	14.958377

Note: 'ascending = False' will sort the data frame in descending order

Rank the dataframe

In [93]:
```
# rank the data frame 'data' in descending order based on 'BMI'
# 'method = min' assigns the minimum rank to highest equal value of 'BMI'
df_basic_info['Rank_min'] = df_basic_info['BMI'].rank(ascending = 0, method  = 'mir
```

In [94]:
```
# print the data
df_basic_info
```

Out[94]:

	Age	Weight (in kg)	Height (in m)	BMI	Rank_min
0	45.0	60.0	1.35	32.921811	5.0
1	12.0	43.0	1.21	29.369579	7.0
2	54.0	78.0	1.50	34.666667	4.0
3	26.0	65.0	1.21	44.395875	1.0
4	68.0	50.0	1.32	28.696051	8.0
5	21.0	43.0	1.52	18.611496	19.0
6	10.0	32.0	1.65	11.753903	23.0
7	57.0	34.0	1.61	13.116778	22.0
8	75.0	23.0	1.24	14.958377	20.0
9	32.0	21.0	1.52	9.089335	24.0
10	23.0	53.0	1.50	23.555556	14.0
11	34.0	65.0	1.76	20.983988	18.0
12	55.0	89.0	1.65	32.690542	6.0
13	23.0	45.0	1.75	14.693878	21.0
14	56.0	76.0	1.69	26.609713	10.0
15	67.0	78.0	1.85	22.790358	15.0
16	26.0	65.0	1.21	44.395875	1.0
17	56.0	74.0	1.69	25.909457	13.0
18	67.0	78.0	1.85	22.790358	15.0
19	26.0	65.0	1.21	44.395875	1.0
20	68.0	50.0	1.32	28.696051	8.0
21	56.0	76.0	1.69	26.609713	10.0
22	67.0	78.0	1.85	22.790358	15.0
24	45.0	85.0	1.80	26.300000	12.0

From the above data frame, we can see that 'BMI = 44.395875' is repeating thrice; thus the method = 'min' will assign the minimum rank (=1) to all the three values of BMI. The rank '4' will be assigned to the second largest value of BMI and so on. Thus, there is no rank equal to 2 and 3.

In [113…
```
# method = 'dense' assigns same rank to all the same BMI values
df_basic_info['Rank_densed'] = df_basic_info['BMI'].rank(method = 'dense')
```

In [114…
```
# print the data
df_basic_info
```

Out[114]:

	Age	Weight (in kg)	Height (in m)	BMI	Rank_min	Rank_densed
0	45.0	60.0	1.35	32.921811	5.0	16.0
1	12.0	43.0	1.21	29.369579	7.0	14.0
2	54.0	78.0	1.50	34.666667	4.0	17.0
3	26.0	65.0	1.21	44.395875	1.0	18.0
4	68.0	50.0	1.32	28.696051	8.0	13.0
5	21.0	43.0	1.52	18.611496	19.0	6.0
6	10.0	32.0	1.65	11.753903	23.0	2.0
7	57.0	34.0	1.61	13.116778	22.0	3.0
8	75.0	23.0	1.24	14.958377	20.0	5.0
9	32.0	21.0	1.52	9.089335	24.0	1.0
10	23.0	53.0	1.50	23.555556	14.0	9.0
11	34.0	65.0	1.76	20.983988	18.0	7.0
12	55.0	89.0	1.65	32.690542	6.0	15.0
13	23.0	45.0	1.75	14.693878	21.0	4.0
14	56.0	76.0	1.69	26.609713	10.0	12.0
15	67.0	78.0	1.85	22.790358	15.0	8.0
16	26.0	65.0	1.21	44.395875	1.0	18.0
17	56.0	74.0	1.69	25.909457	13.0	10.0
18	67.0	78.0	1.85	22.790358	15.0	8.0
19	26.0	65.0	1.21	44.395875	1.0	18.0
20	68.0	50.0	1.32	28.696051	8.0	13.0
21	56.0	76.0	1.69	26.609713	10.0	12.0
22	67.0	78.0	1.85	22.790358	15.0	8.0
24	45.0	85.0	1.80	26.300000	12.0	11.0

Here, the dense method assigns minimum rank (=1) to the minimum value (=9.089335) of the BMI. Rank 2 will be assigned to next smallest value and so on. The value 44.395875 which repeats thrice has the same rank - 18 to the three observations.

To check for missing values

We shall import a new dataset.

In [115…
```
# read the dataset
df_basic_info_missing = pd.read_csv("basic_info_missingdata.csv")
```

In [116…
```
# print the dataset
df_basic_info_missing
```

Out[116]:

	Age	Weight (in kg)	Height (in m)
0	45.0	60.0	1.35
1	12.0	43.0	1.21
2	54.0	78.0	1.50
3	26.0	65.0	1.21
4	68.0	50.0	1.32
5	21.0	NaN	1.52
6	10.0	32.0	1.65
7	57.0	34.0	1.61
8	75.0	23.0	1.24
9	32.0	21.0	1.52
10	23.0	NaN	1.50
11	34.0	65.0	1.76
12	55.0	89.0	1.65
13	23.0	45.0	1.75
14	56.0	76.0	1.69
15	67.0	78.0	1.85
16	26.0	65.0	1.21
17	56.0	74.0	1.69
18	67.0	78.0	NaN
19	26.0	65.0	1.21
20	68.0	50.0	1.32
21	NaN	76.0	1.69
22	67.0	78.0	1.85

Check for missing values

In [100…
```
# isnull() returns True for a missing value
# sum() gives the sum of True values
df_basic_info_missing.isnull().sum()
```

Out[100]:
```
Age              1
Weight (in kg)   2
Height (in m)    1
dtype: int64
```

The function `.isnull` check whether the data is missing. The `sum()` sums the number of 'True' values in the column. The final output gives the number of missing values in each column.

Here, we see there are 2 missing values in the 'weight' column and one missing value in other columns.

In [ ]: