UNIT - 5
WORKING WITH NUMPY

LEARNING OBJECTIVES

After reading this lesson student should be able to understand:

- The concept of Arrays and how they are different from List.
- How multi-dimensional arrays can be created using NumPy built-in functions.
- Accessing individual elements of the array using indexing operation
- Assigning values to individual elements of the array using indexing operation.
- Accessing subset of the array using slicing operation.
- Assigning values to subset of array using slicing operation.
- Executing basic mathematics as well as linear algebra operations using NumPy.
- Why arrays are preferred over Lists.

INTRODUCTION

NumPy stands for Numerical Python, is a fundamental library for scientific or numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.

Some key features and functionalities of NumPy library are:

- Multi-dimensional Arrays
- Vectorized Operations
- Mathematical Functions
- Array Manipulation
- Broadcasting
- Random Number Generation
- Integration with other Libraries

NUMPY ARRAY FUNCTIONS

One of the key features of NumPy is its array object. NumPy allow creating 1-dimensional, 2-dimensional and higher dimensional homogeneous arrays, called n-d array. N-d array is a fast and flexible container for large datasets in Python. Array enables to perform mathematical operations on complete blocks of data using the same syntax that is used to perform operations between scalar elements (such as int, float, strings).

Various functions that can be used to create arrays in NumPy are as follows:

Function Name	**Description**
array( )	Convert input sequence (such as list, tuple, array or any other sequence type) to an n-d array. It creates a copy of the data, by default.
asarray( )	Similar to array( ) function, convert input sequence (such as list, tuple, array or any other sequence type) to an n-d array. However, do not create a copy of the input, if input sequence is already an array.
arange( )	Similar to built-in range( ) function, but creates an n-d array (instead of a list) of elements specified in between start and end value, given as arguments in arange( ) function.
ones( )	Creates an array of all 1s with the given shape and dtype.
ones_like( )	It accepts an array (created using ones( ) function) as input and produces an ones array of the same shape and dtype.
zeros( )	Creates an array of all 0s with the given shape and dtype.
zeros_like ( )	It accepts an array (created using zeros( ) function) as input and produces a zeros array of the same shape and dtype.
empty( )	Create new array but populate it with random or garbage values.
empty_like ( )	It accepts an array (created using empty( ) function) as input and produces a new array of the same shape and dtype.
full( )	Creates an array of the given shape and dtype having all the values specified as “fill value” in the argument.
full_like( )	It accepts an array (created using full( ) function) as input and produces another array of the same shape and dtype.

eye( )	Creates a square *n*x*n* identity matrix (1s on the diagonal and 0s elsewhere).
identity( )	Similar to eye( ) function, creates a square *n*x*n* identity matrix.
random( )	Creates an array of random numbers.
linspace( )	Creates an array of fixed length within specified start and end values.
tile( )	Creates a new array by repeating an existing array for a particular number of times.
randint( )	Creates a random array of integers within a particular range.

CREATING N-D ARRAY

In the previous section, we saw that numerous functions are available in NumPy to create n-d array. How each of these functions is used, we will see in this section.

To use these functions, first of all we need to import the NumPy library. The syntax for the same is as follows:

```
import numpy as np
```

Here, np is an alias name given to the numpy library. This practice is followed because for using any function that is defined under a certain library, it is required to attach the name of the library before the function name. Using a short alias name makes it easier to write the code for programmers.

np.array( ):

First function given in the previous section to create an array is array( ). To use this function for creating n-d array, we first need to define a sequence that can be used as an argument to the array( ) function.

```
# Creating a list 'data1'

data1 = [6,7.5, 8, 0 ,1]
```

Here, the line written after '#' symbol is a comment. That's how single line comments are written in Python. For multiline comments, triple quotes are used.

data1 is a list containing elements of both int and float types.

```
# Using array method to convert list into numpy array

arr1 = np.array(data1)
```

np.array( ) function will accepts ‘data1’ as argument and convert it into an array. On printing the arr1 variable, the output will be as follows:

```
array([6. , 7.5, 8. , 0. , 1. ])
```

As NumPy defines array of homogeneous elements, all the integer list values are converted in to float values.

The array created above is one-dimensional array. It is also possible to create two-dimensional array using list and np.array( ) function. For that, create a nested list and pass it as argument to np.array( ) function.

```
# Creating nested list 'data2'

data2 = [[1,2,3,4], [5,6,7,8]]

# Converting nested list into two-dimensional array 'arr2'

arr2 = np.array(data2)
```

It is also possible to directly pass a nested list to np.array( ) function. It will produce the same result.

```
# Another Way of creating two-dimensional array

arr3 = np.array([[10,20,30,40],[50,60,70,80]])
```

On printing the variables ‘arr2’ and ‘arr3’, the result will be:

```
# Printing two-dimensional array 'arr2'

arr2

 array([[1, 2, 3, 4],
        [5, 6, 7, 8]])


# Printing two-dimensional array 'arr3'

arr3

array([[10, 20, 30, 40],
       [50, 60, 70, 80]])
```

np.asarray( ):

As defined in the previous section, np.asarray( ) function converts input sequence (such as list, tuple, array or any other sequence type) to an n-d array.

Call the np.asarray( ) function and pass the previously created array ‘arr3’ as the argument.

```
arr4 = np.asarray(arr3)
```

Print the newly created array ‘arr4’ as:

```
print(arr4)
```

The output will be:

```
[[10 20 30 40]
 [50 60 70 80]]
```

Here, np.asarray( ) function has created a new array by copying the values from previously created array.

However, it is also possible to create an array using this function by passing a sequence such as list or tuple as an argument.

```
arr5 = np.asarray([11,22,33,44,55])
```

On printing the newly created array as:

```
print(arr5)
```

The output will be:

```
[11 22 33 44 55]
```

np.arange( ):

np.arange( ) function creates an n-d array (instead of a list) of elements specified in between start and end value, given as arguments in np.arange( ) function.

Other than start and end value, the function also use step value as argument:

np.arange(start, end, step)

These three arguments signifies:

- *start:* It specifies the first element value of the array.
- *end:* It specifies the last element value of the array. However, the last value is always *end-1*.
- *step:* It specifies the gap or jump between values. Or it can be defined as the difference between next value and the current value of the array.

Out of these 3 arguments, only end argument is mandatory. The other two arguments, start and step are optional. If you do not specify any value for start and step, then these are initialized by their default values, 0 and 1, respectively.

The example below creates an array ‘arr6’ by using np.arange( ) function having just one mandatory argument ‘end’.

```
arr6 = np.arange(12)
print(arr6)
```

The output of the above print statement will be as follows:

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

An array of 12 elements 0 to 11 is created with the default step value 1.

np.ones( ):

Creates an array of all 1s with the given shape and dtype.

```
arr8 = np.ones(5)
print(arr8)
```

The above statements will create and print a one-dimensional array of 5 elements and all elements will have the value as 1.

```
[1. 1. 1. 1. 1.]
```

By default, the data type of the array is float. If you want to change this data type, the dtype argument can be used while creating the array.

```
arr10 = np.ones((3,4), dtype=int)
print(arr10)
```

This will create a two-dimensional and print an array of 3 rows and 4 columns, filled with 1s and the data type of the array will be integer, as specified with dtype argument.

The output of the print statement will be as follows:

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

np.ones_like( ):

It accepts an array (created using ones( ) function) as input and produces an ones array of the same shape and dtype.

The example below creates an ones array ‘arr12’ by passing an existing ones array ‘arr10’ as argument.

```
arr12 = np.ones_like(arr10)
print(arr12)
```

The output of the print statement will be as follows:

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

You must have noticed that we did not specify the shape and data type of the array in np.ones_like( ) function. But it copied these values from arr10 specified as argument in the function.

np.zeros( ):

Creates an array of all 0s with the given shape and dtype. Here, 'arr13', a one-dimensional array is created having all 0s of integer type.

```
arr13 = np.zeros(5, dtype=int)
print(arr13)
```

The output of print command will be:

```
[0 0 0 0 0]
```

Similar to one-dimensional array, two or higher dimensional zeros array can be created by using np.zeros( ) function.

This statement creates and print a two-dimensional zeros array having 4 rows and 4 columns.

```
arr14 = np.zeros((4,4))
print(arr14)
```

The output of the print statement will be:

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

np.zeros_like( ):

It accepts an array (created using np.zeros( ) function) as input and produces a zeros array of the same shape and dtype. Here, 'arr15' is a zeros array created using 'arr13'. It automatically copies the data type and shape of the array from *arr13*.

```
arr15 = np.zeros_like(arr13)
print(arr15)
```

```
[0 0 0 0 0]
```

Along with copying an existing array, it is also possible to change its shape i.e., changing number of rows and columns.

The statement below creates a zeros array 'arr16' using an existing array 'arr14'. However, instead of using the shape of arr14 (4 rows and 4 columns), it also redefines the shape of newly created array as 2 rows and 8 columns by using reshape( ) function.

```
arr16 = np.zeros_like(arr14).reshape(2,8)
print(arr16)
```

The output of the print statement will be:

```
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

We shall discuss in detail about the reshape( ) function in section 4.7.

np.empty( ):

Create new array with the given shape but populate (initialize) it with random or garbage values. This statement will create and print an one-dimensional array 'arr17' having 10 elements and data type of the array will be integer as specified with dtype argument.

```
arr17 = np.empty(10, dtype=int)
print(arr17)
```

The output of print command will be:

```
[                  0                  37                   0
                   0        416611827712 4189022123868430971
 8391735957915902496 8392569455068064802 3539883574597532194
         537448569949]
```

In the output, you can expect any random value.

Similar to one-dimensional array, higher dimensional array can be created. The statement below creates and print a two-dimensional array of 3 rows and 5 columns, populated with values with float data type.

```
arr18 = np.empty((3,5))
print(arr18)
```

The output will be:

```
[[0.         0.         0.4472136  0.0531494  0.18257419]
 [0.4472136  0.2125976  0.36514837 0.4472136  0.4783446 ]
 [0.54772256 0.4472136  0.85039041 0.73029674 0.4472136 ]]
```

np.empty_like( ):

It accepts an array (created using np.empty( ) function) as input and produces a new array of the same shape and dtype.

The statement below creates and print an array similar to 'arr17' (a one-dimensional array having 10 elements).

```
arr19 = np.empty_like(arr17)
print(arr19)
```

```
[          0 210453399547 197568495664 433791697010 446676598899
 481036337249 171798691941 188978561078 176093659186            0]
```

The np.empty_like( ) function can also be used to create an array similar to any other type of array (an array not created using np.empty( ) function). For example, the statement below creates and print an array similar to 'arr14' (a zeros array having 4 rows and 4 columns).

```
arr20 = np.empty_like(arr14)
print(arr20)
```

The output will be:

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

As stated above, the array created with np.empty( ) function is populated with random or garbage values; it is possible to change the array values using the indexing operation.

To change all the values of 'arr19' to 30, following statement can be used. This statement is simply assigning the value '30' to all the index positions of the array.

```
arr19[:] = 30
print(arr19)
```

The output of the print statement will be:

```
[30 30 30 30 30 30 30 30 30 30]
```

Instead of random values, all values of the array are now 30.

It is also possible to change a specific value of the array. For example, if you want to change the first value from 30 to 40, then following statement can be used:

```
arr19[0] = 40
print(arr19)
```

The index of the array always start from 0. So, first value will have the index 0. By using the index operation, the value '40' can be assigned to index 0.

The output of the above print statement will be:

```
[40 30 30 30 30 30 30 30 30 30]
```

Only the first value will be changed to 40. Rest of the values will remain as 30.

We shall discuss in detail about indexing in the upcoming section 4.6.

np.full( ):

Creates an array of the given shape and dtype having all the values specified as "fill value" in the argument.

This statement will create and print a one-dimensional array 'arr21' having 10 elements and all the values will be 7. Here, np.full( ) function accepts two arguments; first is the number of elements and second is the fill value.

```
arr21 = np.full(10, 7)
print(arr21)
```

The output of print statement will be:

```
[7 7 7 7 7 7 7 7 7 7]
```

The next example creates and print a two-dimensional array 'arr22' having 10 rows and 5 columns and fill value as 7.4.

```
arr22 = np.full((10,5), 7.4)
print(arr22)
```

```
[[7.4 7.4 7.4 7.4 7.4]
 [7.4 7.4 7.4 7.4 7.4]
 [7.4 7.4 7.4 7.4 7.4]
 [7.4 7.4 7.4 7.4 7.4]
 [7.4 7.4 7.4 7.4 7.4]
 [7.4 7.4 7.4 7.4 7.4]
 [7.4 7.4 7.4 7.4 7.4]
 [7.4 7.4 7.4 7.4 7.4]
 [7.4 7.4 7.4 7.4 7.4]
 [7.4 7.4 7.4 7.4 7.4]]
```

np.full_like( ):

It accepts an array (created using np.full( ) function) as input and produces another array of the same shape and dtype.

This statement below creates and print an array 'arr23' having same shape and data type as of array 'arr21' but with the fill value as 15.

```
arr23 = np.full_like(arr21, 15)
print(arr23)
```

The output will be:

```
[15 15 15 15 15 15 15 15 15 15]
```

You must have noticed that in case of ones_like( ), zeros_like( ) and empty_like( ) function, the arrays were created using existing array's

name as argument. However, this is not the case with full_like( ) function. It is mandatory to specify the fill value along with already existing array name as argument, even if you do not want to change the fill value.

If you do not provide the fill value, then this will lead to an error message.

```
arr23 = np.full_like(arr21)
print(arr23)
```

The output of the above print statement will be a TypeError message stating that one positional argument 'fill_value' is missing:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/71/9p4hsdf14knb5ss7tsgbyd7m0000gn/T/ipykernel_3872/235823564.py in <module>
----> 1 arr23 = np.full_like(arr21)
      2 print(arr23)

~/opt/anaconda3/lib/python3.9/site-packages/numpy/core/overrides.py in full_like(*args, **kwargs)

TypeError: _full_like_dispatcher() missing 1 required positional argument: 'fill_value'
```

np.eye( ):

Creates a square *n*x*n* identity matrix (1s on the diagonal and 0s elsewhere).

The statements below create and print an array 'arr25' which is an identity matrix having all float values.

```
arr25 = np.eye(5)
print(arr25)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

Although the identity matrix created above is a two-dimensional matrix, but only one value as an argument is sufficient because it is an identity matrix (having equal number of rows and columns). However, if the values for both rows and columns are given as argument, it produces the same result.

```
arr26 = np.eye(5,5)
print(arr26)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

np.identity( ):

Similar to np.eye( ) function, it creates a square *n*x*n* identity matrix (1s on the diagonal and 0s elsewhere).

The statements below create and print an array 'arr27' which is an identity matrix having all float values.

```
arr27 = np.identity(6)
print(arr27)
```

and the output will be:

```
[[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]]
```

However, unlike np.eye( ) function, np.identity( ) does not accept two values as argument. If you will try to execute the statement given below, it will lead to an error message.

```
arr29 = np.identity(6,6)
print(arr29)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/71/9p4hsdf14knb5ss7tsgbyd7m0000gn/T/ipykernel_3872/2469318804.py in <module>
----> 1 arr29 = np.identity(6,6)
      2 print(arr29)

~/opt/anaconda3/lib/python3.9/site-packages/numpy/core/numeric.py in identity(n, dtype, like)
   2180
   2181     from numpy import eye
-> 2182     return eye(n, dtype=dtype, like=like)
   2183
   2184

~/opt/anaconda3/lib/python3.9/site-packages/numpy/lib/twodim_base.py in eye(N, M, k, dtype, order, like)
    213     if M is None:
    214         M = N
--> 215     m = zeros((N, M), dtype=dtype, order=order)
    216     if k >= M:
    217         return m

TypeError: Cannot interpret '6' as a data type
```

np.random( ):

Creates an array of random numbers. The np.random( ) function is defined in random library. Hence, to call this function, name of random library needs to be attached.

The statements below create and print a two-dimensional array of 3 rows and 4 columns having random numbers in between 0 and 1.

```
arr30 = np.random.random((3, 4))
print(arr30)
```

The output of the print statement will be:

```
[[0.80985837, 0.95242384, 0.91530801, 0.79987372],
 [0.28679703, 0.55929882, 0.82068279, 0.15327797],
 [0.11424299, 0.6713584 , 0.71510406, 0.19358351]]
```

np.randint( ):

Creates a random array of integers within a particular range. The np.randint( ) function accepts three arguments; first one is the start value of range and second argument is the end value of range and third argument specifies that how many numbers need to be generated i.e. the elements of the array.

The statements below create and print a one-dimensional array having 4 elements (specified as last argument) between integer values 2 and 10 (specified as first and second arguments)

```
arr31 = np.random.randint(2, 10, 4)
print(arr31)
```

The output of the print statement will be:

```
[3, 3, 3, 8]
```

As the function generates random integers (numbers), so on executing this function, your output might be different from the one given above.

np.tile( ):

Creates a new array by repeating an existing array for a particular number of times. This is similar to using the multiplication operator with strings.

The statements given below creates and print the array ‘arr32’ by repeating it 3 times using np.tile( ) function.

```
arr32 = ([0, 1, 2])
np.tile(arr32, 3)
print(arr32)
```

The output will be, the array elements “0,1,2” will be repeated 3 times.

```
[0, 1, 2, 0, 1, 2, 0, 1, 2]
```

It is also possible to create two-dimensional array using np.tile( ) function. For this, the function accepts two arguments, first argument is the name of the array being repeated and second argument is the number of rows and number of times the array needs to be repeated in each row.

The statements below create and print a two-dimensional array ‘arr33’ that will contain 3 rows and in each row ‘arr32’ will be repeated 2 times.

```
arr33 = np.tile(arr32, (3,2))
print(arr33)
```

The output of the print statement will be:

```
[[0, 1, 2, 0, 1, 2],
 [0, 1, 2, 0, 1, 2],
 [0, 1, 2, 0, 1, 2]]
```

np.linspace( ):

Creates an array of fixed length within specified start and end values. The np.linspace( ) function accepts three arguments. First argument specifies the start value, second argument specifies the end value and third argument specifies the total number of elements of the array to be created.

This statement will create a one-dimensional array having 15 as start value, 18 as end value and the array will contain 25 elements. The step value between each value is automatically determined by np.linspace( ) function.

```
arr34 = np.linspace(15, 18, 25)
```

```
arr34
```

On printing the array, the output will be:

```
array([15.   , 15.125, 15.25 , 15.375, 15.5  , 15.625, 15.75 , 15.875,
       16.   , 16.125, 16.25 , 16.375, 16.5  , 16.625, 16.75 , 16.875,
       17.   , 17.125, 17.25 , 17.375, 17.5  , 17.625, 17.75 , 17.875,
       18.   ])
```

Help Function

help( ) is a very useful Python function to get information about any built-in function. To find that how a function should be called (i.e., syntax of the function, number of parameters, the kind of values accepted by parameters), the name of the function can be passed as argument while calling the help( ) function.

For example, if user is not aware what np.ones( ) function do and how to call it, then help( ) function can be called as:

```
help(np.ones)
```

The output of this statement will be the detailed description about np.ones( ) function.

```
Help on function ones in module numpy:

ones(shape, dtype=None, order='C', *, like=None)
    Return a new array of given shape and type, filled with ones.

    Parameters
    ----------
    shape : int or sequence of ints
        Shape of the new array, e.g., ``(2, 3)`` or ``2``.
    dtype : data-type, optional
        The desired data-type for the array, e.g., `numpy.int8`.  Default is
        `numpy.float64`.
    order : {'C', 'F'}, optional, default: C
        Whether to store multi-dimensional data in row-major
        (C-style) or column-major (Fortran-style) order in
        memory.
    like : array_like
        Reference object to allow the creation of arrays which are not
        NumPy arrays. If an array-like passed in as ``like`` supports
        the ``__array_function__`` protocol, the result will be defined
        by it. In this case, it ensures the creation of an array object
        compatible with that passed in via this argument.

        .. note::
            The ``like`` keyword is an experimental feature pending on
            acceptance of :ref:`NEP 35 <NEP35>`.

        .. versionadded:: 1.20.0

    Returns
    -------
    out : ndarray
        Array of ones with the given shape, dtype, and order.
```

INSPECTING THE STRUCTURE OF THE ARRAY

Many times it is required to inspect (or check) the structure of the array, especially while working with the large arrays. Some attributes of NumPy that helps in checking the structure of the array are:

(i) *shape:* tells the shape of the array i.e., how many rows and columns the array has.

(ii) *dtype:* tells the data type of the array.

(iii) *ndim:* tells the number of dimensions of the array.

(iv) *itemsize:* tells the number of bytes that each element of the array has occupied in memory.

For example, consider a large size array of size 1000 X 300. Initialize this array with random values as:

```
# Initialising a random 1000 x 300 array

rand_array = np.random.random((1000, 300))
```

Now, the structure of the array can be inspected by using shape, dtype, ndim and itemsize attributes as follows:

```
# Inspecting shape, dtype, ndim and itemsize

print("Shape", rand_array.shape)
print("dtype: ", rand_array.dtype)
print("Dimensions: ", rand_array.ndim)
print("Item size: ", rand_array.itemsize)
```

The output of these statements will be:

```
Shape (1000, 300)
dtype:  float64
Dimensions:  2
Item size:  8
```

First line of the output tells that the array contains 1000 rows and 300 columns.

Second line tells about the data type of the elements of the array i.e., float64. 64 here determines the number of bits required to store each float value in memory.

Third line specifies the number of dimensions. As the array is two-dimensional, hence the output says dimensions as 2.

The last line of the output specifies number of bytes required in memory, to store each element of the array.

SLICING, INDEXING AND ITERATE THROUGH ARRAYS

Indexing:

The operations of accessing a specific element of the array or changing a specific value of the array is called indexing. The array elements are stored at continuous memory locations and can be accessed by their index values.

The index value always starts from 0 and the last index is always the length of the array minus 1 (in case of one-dimensional array). And Python also supports negative indexes that are accessed in reverse direction.

So, for a one-dimensional array having 5 elements [10, 20, 30, 40, 50], the index values will be as:

-5	**-4**	**-3**	**-2**	**-1**	**Negative Indexes**
10	20	30	40	50	**Elements**
0	**1**	**2**	**3**	**4**	**Positive Indexes**

In case of two dimensional array, index values are defines as:

	0	1	2	3
0	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3
3	3,0	3,1	3,2	3,3

The first element points to first row and first column, thus having the index as 0,0. The second element points to first row but second column, thus having the index as 0,1 and so on.

Slicing:

Sometimes, it is required to access the specific elements of the array or we can say to access a subset of the array or to assign the same value to more than one index position. This operation is called slicing the array. Slicing is similar to indexing with the only difference that in slicing, we operate on more than one index value.

Let's understand indexing and slicing operation with the help of some examples.

Consider the following one-dimensional array created using np.arange( ) function:

```
array_1d = np.arange(1, 11)
print(array_1d)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
```

To access any element of the array index values are written in between square brackets.

Third element of the array array_1d can be accessed as:

```
print(array_1d[2])
```

As indexes starts from 0, so the index for the third element will be 2.

and output will be:

```
3
```

Specific indexes can also be provides as list:

```
print(array_1d[[2, 5, 6]])
```

Here a list having 3 index values [2, 3, 5] has been given in between square brackets. The output will be the elements at index positions 2, 3 and 5.

```
[3 6 7]
```

In slicing, we can specify start index, end index and step value as:

array_name[start index: end index: step value]

However, the end index is never included in the output. The last value of output is always *end index – 1*.

During slicing, all these index values (start index, end index and step value) are optional. If start index is not given, by default it is taken as first index i.e., 0. If end index is not given then by default it is taken as *end index + 1* and if step value is not given then it is taken as 1. So, in such scenario, the slicing operation is written as:

array_name[::]

To slice from third element onwards in array 'array_1d':

```
print(array_1d[2:])
```

Here the end index will be taken as *end index + 1* i.e., *9 + 1 =10.* And output will be:

```
[ 3  4  5  6  7  8  9 10]
```

To slice first three elements of array 'array_1d':

```
print(array_1d[:3])
```

Here, the start index will be taken as 0. And output will be:

```
[1 2 3]
```

To slice from third to seventh element in array 'array_1d':

```
print(array_1d[2:7])
```

```
[3 4 5 6 7]
```

To slice from first index till last index but with index increment of 2 in array 'array_1d':

```
print(array_1d[0::2])
```

Here the end index will be taken as *end index + 1* i.e., *9 + 1 =10.* And output will be:

```
[1 3 5 7 9]
```

Indexing and slicing in one-dimensional array is similar to indexing and slicing in the list. However, when these operations are performed in two or higher-dimensional arrays, then index values for rows and columns both need to be specified.

Consider the following two-dimensional array 'array_2d' created using nested list:

```
array_2d = np.array([[2, 5, 7, 5], [4, 6, 8, 10], [10, 12, 15, 19]])
print(array_2d)
```

```
[[ 2  5  7  5]
 [ 4  6  8 10]
 [10 12 15 19]]
```

To access the third row and second column, the indexing can be done as:

```
print(array_2d[2, 1])
```

Here, the index of the row and column will be given separated by comma and the output will be:

```
12
```

To slice second row and all columns:

```
print(array_2d[1, :])
```

The way slice operation can have start index, end index and step value in one-dimensional array, similarly it can have it in two or higher dimensional array also.

In the statement above, to access all columns, only: (colon) is specified which means the column values will be accessed from start index *0* till *end index + 1* but only for second row and the output will be:

```
[ 4  6  8 10]
```

To slice all rows and third column:

```
print(array_2d[:, 2])
```

```
[ 7  8 15]
```

To slice all rows and first three columns:

```
print(array_2d[:, :3])
```

```
[[ 2  5  7]
 [ 4  6  8]
 [10 12 15]]
```

Iterate through Arrays:

Iteration through arrays mean accessing array values using loops. Iterating through one-dimensional array is similar to iteration through list.

Consider the one-dimensional array 'array_1d', created above. To print the squares of all the array values, for loop can be used as:

```
for i in array_1d:
    print(i**2)
```

One by one values will be taken from array_1d and their square will be calculated and printed and the output will be:

```
1
4
9
16
25
36
49
64
81
100
```

However, iteration over two-dimensional arrays is done with respect to first axis i.e., the row. The second axis is column.

Consider the two-dimensional array 'array_2d', created above. To iterate over this array, for loop can be used as:

```
for row in array_2d:
    print(row)
```

This loop will access the rows from array_2d one by one and print it and the output will be:

```
[2 5 7 5]
[ 4  6  8 10]
[10 12 15 19]
```

CHANGING THE SHAPE AND COMBINING ARRAYS

Changing the Shape of Array:

As discussed in the previous section, it is possible to change the shape of the array i.e., the number of rows and columns can be changed. To reshape the array, *reshape( )* function is used.

reshape( ) function can be used while creating the array or on already existing array. However, in both the cases, the array should be reshaped

in such a way that the number of elements remain same, otherwise the error message will be generated.

To change the shape of the array while creating it, the reshape( ) function can be used as:

```
some_array = np.arange(0, 12).reshape(3, 4)
print(some_array)
```

The arange( ) function creates one-dimensional array that can be converted into two-dimensional array by using the reshape function. The output of the above print statement will be a two-dimensional array having 3 rows and 4 columns.

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

The above array can be further reshaped as:

```
some_array.reshape(2, 6)
```

The output will be an array having 2 rows and 6 columns:

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

If you are not sure about the second dimension, then -1 can be specified as second dimension and Python automatically calculates it according to the first dimension and number of elements.

```
some_array.reshape(4, -1)
```

Here, -1 means whatever dimension is needed. The output will be an array having 4 rows and 3 columns.

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Combining Arrays:

Arrays can be combined horizontally as well as vertically. This operation is called Stacking arrays. To combine arrays horizontally, *np.hstack( )* function is used and to combine arrays vertically, *np.vstack( )* function is used.

For horizontal stacking, the number of rows should be the same, while for vertical stacking, the number of columns should be the same.

Consider the following arrays *array_1*, *array_2* and *array_3*, created using np.arange( ) function:

```
array_1 = np.arange(12).reshape(3, 4)
array_2 = np.arange(20).reshape(5, 4)
array_3 = np.arange(15).reshape(3, 5)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

Vertical stacking of array_1 (3 rows, 4 columns) and array_2 (5 rows, 4 columns) can be done as:

```
np.vstack((array_1, array_2))
```

The output of the above statement will be:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

Horizontal stacking of array_1 (3 rows, 4 columns) and array_3 (3 rows, 5 columns) can be done as:

```
np.hstack((array_1, array_3))
```

The output of the above merging will be:

```
array([[ 0,  1,  2,  3,  0,  1,  2,  3,  4],
       [ 4,  5,  6,  7,  5,  6,  7,  8,  9],
       [ 8,  9, 10, 11, 10, 11, 12, 13, 14]])
```

MATHEMATICAL OPERATIONS ON NUMPY ARRAY

NumPy supports various basic as well as Linear Algebra mathematical operations. The mathematical functions are applied to each element of the array.

Applying Basic Mathematical Operations using NumPy:

Consider a one-dimensional array ‘a’, created using np.arange( ) function:

```
a = np.arange(1, 20)
print(a)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Basic mathematical operations sin, cos, exponential and log can be applied as:

```
print(np.sin(a))
print(np.cos(a))
print(np.exp(a))
print(np.log(a))
```

The function will be applied on each element of the array and the output of the print statements will be:

```
[ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427 -0.2794155
  0.6569866   0.98935825  0.41211849 -0.54402111 -0.99999021 -0.53657292
  0.42016704  0.99060736  0.65028784 -0.28790332 -0.96139749 -0.75098725
  0.14987721]



[ 0.54030231 -0.41614684 -0.9899925  -0.65364362  0.28366219  0.96017029
  0.75390225 -0.14550003 -0.91113026 -0.83907153  0.0044257   0.84385396
  0.90744678  0.13673722 -0.75968791 -0.95765948 -0.27516334  0.66031671
  0.98870462]



[2.71828183e+00 7.38905610e+00 2.00855369e+01 5.45981500e+01
 1.48413159e+02 4.03428793e+02 1.09663316e+03 2.98095799e+03
 8.10308393e+03 2.20264658e+04 5.98741417e+04 1.62754791e+05
 4.42413392e+05 1.20260428e+06 3.26901737e+06 8.88611052e+06
 2.41549528e+07 6.56599691e+07 1.78482301e+08]



[0.         0.69314718 1.09861229 1.38629436 1.60943791 1.79175947
 1.94591015 2.07944154 2.19722458 2.30258509 2.39789527 2.48490665
 2.56494936 2.63905733 2.7080502  2.77258872 2.83321334 2.89037176
 2.94443898]
```

Performing Matrix Operations using NumPy:

NumPy allows element wise addition, subtraction, multiplication and division of the matrix. Apart from these basic operations, NumPy also provide function and attribute to perform transpose of a matrix.

Consider the following two matrices ‘mat1’ and ‘mat2’:

```
mat1 = np.arange(1, 13).reshape(3, 4)
mat2 = np.arange(1, 13).reshape(3, 4)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

The basic mathematical operations between these two matrices can be performed similar way it is performed between scalar types.

Addition can be done as:

```
addition = mat1 + mat2

print(addition)
```

Each element of mat1 will be added with the same index element of mat2 and the output of element-wise matrix addition will be:

```
[[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]]
```

Subtraction can be done as:

```
subtraction = mat1 - mat2

print(subtraction)
```

```
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

Multiplication can be done as:

```
multiplication = mat1 * mat2

print(multiplication)
```

```
[[  1   4   9  16]
 [ 25  36  49  64]
 [ 81 100 121 144]]
```

Division can be done as:

```
division = mat1 / mat2

print(division)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Transpose of a matrix can be done by either using *'T'* attribute or using *transpose( )* function.

Consider the following matrix 'mat3':

```
mat3 = np.arange(16).reshape((4,4))
```

```
print(mat3)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

Transpose of this matrix can be calculated as:

```
mat3.T
```

And the output will be:

```
array([[ 0,  4,  8, 12],
       [ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15]])
```

Row values will be swapped to column values and vice-versa.

Another way of performing transpose of matrix is using the transpose( ) function.

```
mat3.transpose(1,0)
```

In case of transpose( ) function, we need to specify the axis as argument that we want to transpose. In two-dimensional array, rows are represented as axis 0 and columns are represented as axis 1. The above statement specifies in transpose( ) function to swap the axes, keep axis 0 at the place of axis 1 and keep axis 1 at the place of axis 0.

The output after transpose will be:

```
array([[ 0,  4,  8, 12],
       [ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15]])
```

Applying Basic Linear Algebra Operations using NumPy:

NumPy provides *np.linalg* package to apply common linear algebra operations such as inverse of matrix, determinant of a matrix, eigenvalues and eigenvectors of a matrix and matrix multiplication (not element-wise multiplication).

Consider the following matrices *a* and *b*:

```
a = np.arange(1, 10).reshape(3, 3)
b= np.arange(1, 13).reshape(3, 4)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

The inverse of the matrix can be calculated using *np.linalg.inv( )* function as:

```
np.linalg.inv(a)
```

The output will be:

```
array([[ 3.15251974e+15, -6.30503948e+15,  3.15251974e+15],
       [-6.30503948e+15,  1.26100790e+16, -6.30503948e+15],
       [ 3.15251974e+15, -6.30503948e+15,  3.15251974e+15]])
```

The determinant of the matrix can be calculated using *np.linalg.det( )* function as:

```
np.linalg.det(a)
```

The output will be:

```
-9.51619735392994e-16
```

The eigen value and eigen vector of the matrix can be calculated using *np.linalg.eig( )* function as:

```
np.linalg.eig(a)
```

The output will be:

```
(array([ 1.61168440e+01, -1.11684397e+00, -3.38433605e-16]),
 array([[-0.23197069, -0.78583024,  0.40824829],
        [-0.52532209, -0.08675134, -0.81649658],
        [-0.8186735 ,  0.61232756,  0.40824829]]))
```

The multiplication of the matrix (also called dot product) can be

calculated using *np.dot( )* function as:

```
np.dot(a, b)
```

The output will be:

```
array([[ 38,  44,  50,  56],
       [ 83,  98, 113, 128],
       [128, 152, 176, 200]])
```

TRADE-OFF BETWEEN ARRAY AND LIST

Using NumPy for Speed:

The key advantages of NumPy library are convenience and speed of computation. Analyst often work with extremely large data sets and thus it is important to understand that how much computational time and memory can be saved using NumPy in comparison to standard Python list.

Let's understand it by comparing the computation time of arrays and list for calculating the element-wise product of numbers.

```
## Comparing time taken for computation

list_1 = [i for i in range(1000000)]
list_2 = [j**2 for j in range(1000000)]

# list multiplication

import time

# store start time, time after computation, and take the difference

t0 = time.time()

product_list = []
for i in range(1000000):
    product_list.append(list_1[i]*list_2[i])
t1 = time.time()
list_time = t1 - t0
print(t1-t0)


# numpy array

array_1 = np.array(list_1)
array_2 = np.array(list_2)

t0 = time.time()
array_3 = array_1*array_2
t1 = time.time()
numpy_time = t1 - t0

print(t1-t0)

print("The ratio of time taken is {}".format(list_time/numpy_time))
```

The output of the above code will be:

```
0.15524983406066895
0.0020508766174316406
The ratio of time taken is 75.69925598697978
```

The reason for such difference in speed is that NumPy is written in C that executes behind the scene. Also, NumPy arrays are more compact than list i.e., they take much lesser storage space than lists.

SELF-ASSESSMENT QUESTIONS

Q1. Create the following list having multiples of 3:

list1 = [3, 6, 9, 12, 15, 18, 21, 24]

Convert this list into a one-dimensional array.

Q2. Create a one-dimensional array of random integers from 21 to 40. The array should contain 10 elements.

Q3. Create a two-dimensional array of size 5 x 7. All the values of the array should be same. You can pick any value of your choice.

Q4. Convert the one-dimensional array created in question 1 into two-dimensional array of size 2 x 4. Can array be converted into two-dimensional array of any other size?

Q5. Create an identity matrix of size 5 x 5.

Q6. Create a one-dimensional array having 20 elements using empty( ) function. Assign the value '20' to the first half elements of the array and assign value '45' to the second half elements of the array.

Q7. Create a two-dimensional array of 5 x 7 and print following:

print the value of third row and fifth column.

print all the values of fifth row.

print all the values of second column.

print all the rows and first two columns.

print last three rows and all the columns.

Q8. Create a matrix of 4 x 3 and calculate transpose of this matrix.

Q9. Create a matrix of 4 x 4 and find its determinant, eigen value and eigen vector.

Q10. Create 2 two-dimensional arrays of size 4 x 3 and 3 x 3. Merge these arrays vertically.

UNIT - 5
WORKING WITH PANDAS

INTRODUCTION

Python Pandas is a powerful open-source library for data manipulation, analysis and visualization. It is built on the top of the NumPy library and provides high-level data structures and functions to efficiently work with structured data, such as tabular data, time series and heterogenous data.

Some key features and functionalities of Pandas library are:

- Data Frame
- Data Manipulation
- Data Input/Output
- Time Series Analysis
- Handling Missing Data
- Data Visualization
- Integration with other Libraries

PANDAS DATA STRUCTURE

The primary data structures in Pandas are *Series* and *DataFrame*. A series is similar to a one-dimensional NumPy array that contains a sequence of homogeneous values. However, it also consist of an array of data labels (called index), which makes it different from NumPy array.

A DataFrame is a two-dimensional data structure that represents information in tabular format, similar to an Excel sheet or a SQL table. A DataFrame can be seen as a table where each column is a Pandas series. It allows storing and manipulating heterogeneous data efficiently.

WORKING WITH SERIES

As discussed in the previous section, a series is a one-dimensional array-like object that stores the sequence of homogeneous values. Apart, from the sequence of values, it also stores the data labels. To create a series in Pandas, *Series( )* method is used. However, as the method belongs to Pandas library, first it is required to import the library.

The library can be imported as:

```
import pandas as pd
```

Here, pd is the alias name given to Pandas library.

To create a numeric pandas series, the Series( ) method will be used as:

```
s = pd.Series([2, 4, 5, 6, 9])
```

This statement will create a Pandas series *'s'* having sequence of integer values. On printing the variable ***s***, the output will be:

```
0    2
1    4
2    5
3    6
4    9
```

First column of the output represents the index and it is by default a numeric index having integer values that always starts from 0 (zero). The second column represents the series values.

Similar to numeric series, character series can also be created:

```
char_series = pd.Series(['a', 'b', 'af'])
char_series
```

The output of the above statement will be:

```
0     a
1     b
2    af
```

The series of type Datetime can also be created either by specifying the individual dates in the list or by using start and end arguments with the date format as 'MM-DD-YYYY':

```
date_series = pd.date_range(start = '11-09-2017', end = '12-12-2017')
date_series
```

The output of the above statement will be:

```
DatetimeIndex(['2017-11-09', '2017-11-10', '2017-11-11', '2017-11-12',
               '2017-11-13', '2017-11-14', '2017-11-15', '2017-11-16',
               '2017-11-17', '2017-11-18', '2017-11-19', '2017-11-20',
               '2017-11-21', '2017-11-22', '2017-11-23', '2017-11-24',
               '2017-11-25', '2017-11-26', '2017-11-27', '2017-11-28',
               '2017-11-29', '2017-11-30', '2017-12-01', '2017-12-02',
               '2017-12-03', '2017-12-04', '2017-12-05', '2017-12-06',
               '2017-12-07', '2017-12-08', '2017-12-09', '2017-12-10',
               '2017-12-11', '2017-12-12'],
```

To access only values from Pandas series, values property can be used:

```
char_series.values
```

The output of this statement will be:

```
array(['a', 'b', 'af'], dtype=object)
```

To access only index values from Pandas series, index property can be used:

```
char_series.index
```

The output will be:

```
RangeIndex(start=0, stop=3, step=1)
```

As default index is a sequence of integer values starting from 0, so in output, it specifies start value, stop value and step value (difference between each index value). However, the last index value is always stop value minus 1.

Explicitly Specifying Indices

It is also possible to specify the index values explicitly by using the *index* argument while creating the series using *pd.Series( )* method.

The statement below creates a Pandas series having integer values and character value indices.

```
pd.Series([0, 1, 2], index = ['a', 'b', 'c'])
```

The output of this statement will be:

```
a    0
b    1
c    2
```

First column representing the index and second column representing the values.

Another way of specifying explicit indices is with the help of range function. The statement below creates a sequence using range( ) function and converts it into a one-dimensional array using *np.array( )* function (do not forget to import NumPy library before using this function) and calculate square of each value of array. This array is then converted into a series with explicit indexing generated using range( ) function.

```
pd.Series(np.array(range(0,10))**2, index = range(0,10))
```

One thing that needs to be taken care is that the number of elements in the index list and the number of elements specified in the series should be equal.

The output of the above statement will be:

```
0     0
1     1
2     4
3     9
4    16
5    25
6    36
7    49
8    64
9    81
```

Usually we work with series as part of a DataFrame, so let's start the discussion about DataFrame.

WORKING WITH DATA FRAME

DataFrame is the most widely used data structure in data analysis. Data in DataFrame is represented in the form of table having rows and columns, with rows having an index and columns having meaningful names. Each column in the DataFrame can have different value type.

There are various ways of creating DataFrame, such as creating from Dictionary, JSON (JavaScript Object Notation) objects, reading from text, CSV files etc.

Creating DataFrame from Dictionary

While creating the DataFrame from dictionary, the dictionary should contain equal-length list (stored as values associated with keys in dictionary). When DataFrame is created from dictionary, keys of the dictionary become column names.

To create a DataFrame in Pandas, *DataFrame( )* method is used.

```
data = {'name': ['Vinay', 'Kushal', 'Aman', 'Saif'],
                'age': [22, 25, 24, 28],
                'occupation': ['engineer', 'doctor', 'data analyst', 'teacher']}
```

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

The statements written above will create a dictionary *'data'* and convert it into a DataFrame and assign it to the variable 'df' and print it. The output of the above 'df' statement will be:

	name	age	occupation
0	Vinay	22	engineer
1	Kushal	25	doctor
2	Aman	24	data analyst
3	Saif	28	teacher

Like series, a numeric index is created automatically in DataFrame also.

To arrange the columns in specific order in the DataFrame, columns argument can be used. The columns argument specifies the order of the columns to be displayed in the DataFrame.

```
df = pd.DataFrame(data, columns=['name', 'occupation', 'age'])
df
```

The output of the 'df' statement will be:

	name	occupation	age
0	Vinay	engineer	22
1	Kushal	doctor	25
2	Aman	data analyst	24
3	Saif	teacher	28

If a column name that does not exist in dictionary, is specified in columns argument then it is filled with missing values (NaN, Not a Number) in the result.

```
df = pd.DataFrame(data, columns=['name', 'occupation', 'age', 'address'])
df
```

The result will be as follows:

	name	occupation	age	address
0	Vinay	engineer	22	NaN
1	Kushal	doctor	25	NaN
2	Aman	data analyst	24	NaN
3	Saif	teacher	28	NaN

The values in the address column can be filled as:

```
df['address'] = 'Delhi'
df
```

However, it will fill all the values as ‘Delhi’.

	name	occupation	age	address
0	Vinay	engineer	22	Delhi
1	Kushal	doctor	25	Delhi
2	Aman	data analyst	24	Delhi
3	Saif	teacher	28	Delhi

To change the value of a specific cell:

```
df['address'][0] = 'Bombay'
df
```

This will change the address value of first row:

	name	occupation	age	address
0	Vinay	engineer	22	Bombay
1	Kushal	doctor	25	Delhi
2	Aman	data analyst	24	Delhi
3	Saif	teacher	28	Delhi

To set the distinct values in address column for all the rows together, a sequence can be assigned to the column:

```
addr = ['Delhi', 'Bombay', 'Chennai', 'Calcutta']
df['address'] = addr
df
```

The output will be:

	name	age	occupation	address
0	Vinay	22	engineer	Delhi
1	Kushal	25	doctor	Bombay
2	Aman	24	data analyst	Chennai
3	Saif	28	teacher	Calcutta

A column in the DataFrame can be accessed as a series by using the column name as attribute.

```
df.name
```

The output will be:

```
0     Vinay
1    Kushal
2      Aman
3      Saif
```

Changing the Default Numeric Index

The default numeric index of DataFrame can be changed by using index argument:

```
df = pd.DataFrame(data, index=['one', 'two', 'three', 'four'])
df
```

The output will be:

	name	age	occupation
one	Vinay	22	engineer
two	Kushal	25	doctor
three	Aman	24	data analyst
four	Saif	28	teacher

REINDEXING

Pandas provide *reindex( )* method that creates a new object and associate the data with the new index.

Consider the **data** dictionary having name, age and occupation of employees of an organization.

```
data = {'name': ['Vinay', 'Kushal', 'Aman', 'Saif'],
                'age': [22, 25, 24, 28],
                'occupation': ['engineer', 'doctor', 'data analyst', 'teacher']}
```

The dictionary will be converted into a DataFrame with the character index:

```
df = pd.DataFrame(data, index=['two', 'one', 'four', 'three'])
df
```

The output of **df** statement will be:

	name	age	occupation
two	Vinay	22	engineer
one	Kushal	25	doctor
four	Aman	24	data analyst
three	Saif	28	teacher

The rows of the above DataFrame can be reindexed as:

```
df2 = df.reindex(['one', 'two', 'three', 'four', 'five'])
df2
```

This will create a new DataFrame object **df2** and output of **df2** statement will be:

	name	age	occupation
one	Kushal	25.0	doctor
two	Vinay	22.0	engineer
three	Saif	28.0	teacher
four	Aman	24.0	data analyst
five	NaN	NaN	NaN

In the DataFrame **df** there was no such index value **'five'**, therefore no data with respect to name, age and occupation column is associated with this index. However, during reindexing this index will be added in **df2** and the values for all the columns will be initiated as 'NaN'.

Dropping Entries from an Axis

Dropping one or more entries from an axis is easy with the help of an index. Pandas *drop( )* method return a new object containing remaining entries of the DataFrame.

Consider the previously created DataFrame df2. The statement below creates a new DataFrame df3 after dropping the row from df2 having index *'two'*.

```
df3 = df2.drop('two')
df3
```

The output will be a new DataFrame having remaining rows:

	name	age	occupation
one	Kushal	25.0	doctor
three	Saif	28.0	teacher
four	Aman	24.0	data analyst
five	NaN	NaN	NaN

It is also possible to drop more than one row. The index of the rows should be provided as a list as argument.

```
df3 = df2.drop(['four', 'five'])
df3
```

The output will be:

	name	age	occupation
one	Kushal	25.0	doctor
two	Vinay	22.0	engineer
three	Saif	28.0	teacher

Calling drop( ) method with a list of index labels, by default drop rows. However, it is also possible to drop columns by using **axis** argument. The axis argument can accept the value as **'1'** or '**columns'**.

Using axis = 1:

```
df4 = df2.drop('age', axis=1)
df4
```

The output will be:

	name	occupation
one	Kushal	doctor
two	Vinay	engineer
three	Saif	teacher
four	Aman	data analyst
five	NaN	NaN

Using axis = 'columns'

```
df5 = df2.drop(['age', 'occupation'], axis='columns')
df5
```

The output will be:

	name
one	Kushal
two	Vinay
three	Saif
four	Aman
five	NaN

In certain scenario, it is required to drop the rows or columns from the DataFrame without making a copy of it. For such cases, **inplace** argument of *drop( )* method can be used. By setting inplace = True, the modifications will be done in original DataFrame without creating any copy of it.

```
df2.drop('five', inplace=True)
df2
```

The output will be:

	name	age	occupation
one	Kushal	25.0	doctor
two	Vinay	22.0	engineer
three	Saif	28.0	teacher
four	Aman	24.0	data analyst

Accessing Top and Bottom rows from DataFrame

Before performing analysis on the data, sometimes it is required to check the type of data contained by DataFrame. On printing the DataFrame, complete DataFrame is printed. This works well with smaller DataFrame, however, for large DataFrame, accessing only few top or bottom rows are sufficient to get the information about the DataFrame. This can be done by using *head( )* or *tail( )* methods.

The statements below will create a new DataFrame having 7 rows.

```
new_data = {'name': ['Vinay', 'Kushal', 'Aman', 'Saif', 'Anny', 'Riya', 'Pihu'],
        'subject': ['Maths', 'English', 'Chemistry', 'Physics', 'Computer', 'GK', 'Economics'],
        'marks': [22, 25, 24, 28, 23, 20, 27]}
```

```
new_df = pd.DataFrame(new_data)
new_df
```

The output will be:

	name	subject	marks
0	Vinay	Maths	22
1	Kushal	English	25
2	Aman	Chemistry	24
3	Saif	Physics	28
4	Anny	Computer	23
5	Riya	GK	20
6	Pihu	Economics	27

To access first five rows *head( )* method is used as:

```
new_df.head()
```

The output will be:

	name	subject	marks
0	Vinay	Maths	22
1	Kushal	English	25
2	Aman	Chemistry	24
3	Saif	Physics	28
4	Anny	Computer	23

To access last five rows *tail( )* method is used as:

```
new_df.tail()
```

The output will be:

	name	subject	marks
2	Aman	Chemistry	24
3	Saif	Physics	28
4	Anny	Computer	23
5	Riya	GK	20
6	Pihu	Economics	27

To access first three rows *head( )* method with argument as number of rows as integer value is used as:

```
new_df.head(3)
```

The output will be:

	name	subject	marks
0	Vinay	Maths	22
1	Kushal	English	25
2	Aman	Chemistry	24

To access last three rows *tail( )* method with argument as number of rows as integer value is used as:

```
new_df.tail(3)
```

The output will be:

	name	subject	marks
4	Anny	Computer	23
5	Riya	GK	20
6	Pihu	Economics	27

INDEXING, SLICING AND FILTERING

Indexing and Slicing Series

Indexing series is similar to indexing one-dimensional NumPy arrays. Consider the following lists, marks and name:

```
marks = [56, 74, 88, 52, 95]
name = ['Ritu', 'Pihu', 'Meera', 'Aarti', 'Siya']
```

The list marks has been converted into a Pandas series having name as indices:

```
student = pd.Series(marks, index=name)
student
```

The output will be:

```
Ritu     56
Pihu     74
Meera    88
Aarti    52
Siya     95
```

Here, first column is the index of the series and second column contain series values.

To access the third row from the series, either integer or character index values can be used.

Using the character index value:

```
student['Meera']
```

The output will be:

```
88
```

Using the integer index value:

```
student[2]
```

And the output will be:

```
88
```

Both ways results in the same output.

It is also possible to fetch values from multiple index positions. For that the index values need to be passed as a list in the argument.

To access second, third and fourth rows from the series:

```
student[[1,2,3]]
```

The output will be:

```
Pihu     74
Meera    88
Aarti    52
```

Same can be achieved using the character indices:

```
student[['Pihu', 'Meera', 'Aarti']]
```

And the output will be same:

```
Pihu     74
Meera    88
Aarti    52
```

Though indexing is producing the similar result while using the integer or character indices. But it is not true while performing slicing on series.

Slicing using integer indices does not include end value in the result. The statement below will extract only second and third row (rows having

index 1 and 2) only. The fourth row (row having index 3) will not be part of the output.

```
student[1:3]
```

The output will be:

```
Pihu      74
Meera     88
```

However, in case of character index slicing, the end index value row is included in the output. Consider the following statement to extract rows from second to fourth:

```
student['Pihu':'Aarti']
```

The output will be:

```
Pihu      74
Meera     88
Aarti     52
```

Here, fourth row has been included in the output.

Indexing and Slicing DataFrame

Indexing and slicing in DataFrame is done to access one or more columns and rows to retrieve a single value or sequence of values.

Consider the DataFrame **df2**:

	name	age	occupation
one	Kushal	25.0	doctor
two	Vinay	22.0	engineer
three	Saif	28.0	teacher
four	Aman	24.0	data analyst

To access a specific column from DataFrame, the name of the column need to be given in square brackets:

```
df2['name']
```

The output will be:

```
one       Kushal
two        Vinay
three       Saif
four        Aman
```

To access a set of columns, columns names should be given as a list within square bracket:

```
df2[['name', 'age']]
```

The output will be:

	name	age
one	Kushal	25.0
two	Vinay	22.0
three	Saif	28.0
four	Aman	24.0

To access a specific row, either character or integer index can be used:

```
df2['two':'two']
```

or

```
df2[1:2]
```

The output will be:

	name	age	occupation
two	Vinay	22.0	engineer

It is similar to slicing operation, to access set of rows, the start and end values can be given as slicing operation:

```
df2[1:3]
```

The output will be:

	name	age	occupation
two	Vinay	22.0	engineer
three	Saif	28.0	teacher

To access a specific cell value, specify the column name and use the values property with specific row value:

```
df2['name'].values[1]
```

Here, the statement will access the value from **second** row of **name** column and the output will be:

```
'Vinay'
```

Selection with loc and iloc

DataFrame also provide the special indexing operators ***loc*** and ***iloc***. These operators allows selecting subset of row and columns from the DataFrame using NumPy like notation. ***loc*** use axis labels and ***iloc*** use integer index.

The syntax of ***loc*** and ***iloc*** indexing operators is:

df.loc[row range, column range]

Here df is the name of the DataFrame.

To access all columns and specific row, specify the character index label (in case of ***loc***) or integer index label (in case of ***iloc***). To access all the columns, a: (colon) sign will include all the columns in the output.

Using ***loc*** indexing operator:

```
df2.loc['two',:]
```

Using ***iloc*** indexing operator:

```
df2.iloc[1,:]
```

Both statements will produce the same output:

```
name              Vinay
age                22.0
occupation    engineer
```

To access all rows and a specific column:

```
df2.loc[:,'name']
```

or

```
df2.iloc[:,0]
```

The output will be:

```
one       Kushal
two        Vinay
three       Saif
four        Aman
```

To access a specific cell value,

```
df2.loc['two', 'name']
```

or

```
df2.iloc[1,0]
```

The output will be:

```
'Vinay'
```

To access set of rows and columns:

```
df2.loc['one':'three', ['name','age']]
```

or

```
df2.iloc[0:3, 0:2]
```

The output will be:

	name	age
one	Kushal	25.0
two	Vinay	22.0
three	Saif	28.0

FUNCTION APPLICATION

An important operation that is frequently used with DataFrame is to apply a function on a specific row or a specific column. This can be done by using DataFrame's *apply( )* method.

Consider the DataFrame **new_df** that stores the marks of the students:

	name	subject	marks
0	Vinay	Maths	22
1	Kushal	English	25
2	Aman	Chemistry	24
3	Saif	Physics	28
4	Anny	Computer	23
5	Riya	GK	20
6	Pihu	Economics	27

Suppose as a data analyst you want to increase the marks of every students by 10. Then the function can be created using lambda expression and can be applied on the column *marks*.

Define lambda expression to increase the value by 10.

```
f = lambda x: x+10
```

Now, apply the function ***f*** on column *marks,* using the *apply( )* method.

```
new_df['marks'].apply(f)
```

The above statement will apply function f on column marks. This will be done by picking each value of marks column and passing it to

lambda expression which in turn will increase the value by 10.

The output of this statement will be:

```
0    32
1    35
2    34
3    38
4    33
5    30
6    37
```

SORTING INDEX AND VALUES

Sorting

Another important built-in operation of DataFrame is Sorting. Sorting can be done by using indices or by using values. To sort the rows or columns using index, *sort_index( )* method is used and to sort row or column using values, *sort_values( )* method is used.

Consider the DataFrame **new_df** that stores the marks of the students:

	name	subject	marks
0	Vinay	Maths	22
1	Kushal	English	25
2	Aman	Chemistry	24
3	Saif	Physics	28
4	Anny	Computer	23
5	Riya	GK	20
6	Pihu	Economics	27

Let's reindex the DataFrame by making column *'name'* as index. For this *set_index( )* method can be used.

```
new_df1 = new_df.set_index('name')
```

The output will be:

name	subject	marks
Vinay	Maths	22
Kushal	English	25
Aman	Chemistry	24
Saif	Physics	28
Anny	Computer	23
Riya	GK	20
Pihu	Economics	27

Now, to sort the index of new_df1 DataFrame, execute following statement:

```
new_df1.sort_index()
```

The output will be:

name	subject	marks
Aman	Chemistry	24
Anny	Computer	23
Kushal	English	25
Pihu	Economics	27
Riya	GK	20
Saif	Physics	28
Vinay	Maths	22

By default, the index labels will be sorted in ascending order. To sort it in descending order, ascending argument need to set as False.

```
new_df1.sort_index(ascending=False)
```

The output will be:

name	subject	marks
Vinay	Maths	22
Saif	Physics	28
Riya	GK	20
Pihu	Economics	27
Kushal	English	25
Anny	Computer	23
Aman	Chemistry	24

To sort the column names, axis argument need to be used:

```
new_df1.sort_index(axis=1)
```

This will sort the names of the columns in the ascending order and the output will be:

name	marks	subject
Vinay	22	Maths
Kushal	25	English
Aman	24	Chemistry
Saif	28	Physics
Anny	23	Computer
Riya	20	GK
Pihu	27	Economics

To sort the DataFrame by values of column or rows, sort_values( ) method is used. The name of the column is specified using ***by*** argument.

The statement below sort the column *marks* in ascending order.

```
new_df1.sort_values(by='marks')
```

And the output will be:

name	subject	marks
Riya	GK	20
Vinay	Maths	22
Anny	Computer	23
Aman	Chemistry	24
Kushal	English	25
Pihu	Economics	27
Saif	Physics	28

To sort the marks column in descending order:

```
new_df1.sort_values(by='marks', ascending=False)
```

The output will be:

name	subject	marks
Saif	Physics	28
Pihu	Economics	27
Kushal	English	25
Aman	Chemistry	24
Anny	Computer	23
Vinay	Maths	22
Riya	GK	20

SUMMARIZING AND COMPUTING DESCRIPTIVE STATISTICS

Summary and Descriptive Statistics

Grouping and aggregation of data are some of the most frequent operations used in data analysis, especially while doing the Exploratory Data Analysis (EDA). A very common task in EDA is comparing the summary statistics across group of data.

To understand what information we can get from descriptive statistics, consider a new DataFrame **df_stat**. The DataFrame consist of two columns comprises of some integer, floating point and NaN values.

```
df_stat = pd.DataFrame({'A':[70.32, np.nan, 30.42, 45, 92, np.nan], 'B':[np.nan, 77.32, 54.32, np.nan, 87, 63]},
                       index=['one', 'two', 'three','four', 'five', 'six'])
```

```
df_stat
```

	A	B
one	70.32	NaN
two	NaN	77.32
three	30.42	54.32
four	45.00	NaN
five	92.00	87.00
six	NaN	63.00

To get the summary statistics, individual methods such as *min( )*, *max( )*, *sum( )*, *mean( )*, *count()* can be used. However, to get the complete statistics in just one statement, *describe( )* method can be used.

Let's see what statistical information these methods provide.

To calculate the min value of a column, *min( )* method can be used:

```
df_stat.min()
```

Output of the above statement will be the minimum value present in each column.

```
A    30.42
B    54.32
```

To calculate the minimum value for each row, axis argument need to be used:

```
df_stat.min(axis='columns')
```

And output will be:

```
one      70.32
two      77.32
three    30.42
four     45.00
five     87.00
six      63.00
```

Similarly *max( ), sum( )*, *count( )*, *mean( )* methods can be used.

Calculating maximum values for columns and rows:

```
df_stat.max()
```

```
A    92.0
B    87.0
```

```
df_stat.max(axis='columns')
```

```
one      70.32
two      77.32
three    54.32
four     45.00
five     92.00
six      63.00
```

Counting values for columns and rows:

```
df_stat.count()
```

count( ) method does not count NaN values, hence the output is 4 for both the columns *A* and *B*.

```
A    4
B    4
```

```
df_stat.count(axis='columns')
```

```
one      1
two      1
three    2
four     1
five     2
six      1
```

Calculating sum of values for columns and rows:

```
df_stat.sum()
```

```
A    237.74
B    281.64
```

```
df_stat.sum(axis='columns')
```

```
one       70.32
two       77.32
three     84.74
four      45.00
five     179.00
six       63.00
```

Calculating mean of values for columns and rows:

```
df_stat.mean()
```

```
A    59.435
B    70.410
```

```
df_stat.mean(axis='columns')
```

```
one      70.32
two      77.32
three    42.37
four     45.00
five     89.50
six      63.00
```

Apart from the statistical information that can be calculated by using above mentioned methods, sometimes it is required to analyse the quartile information of dataset. This complete statistics can be displayed using describe( ) method.

```
df_stat.describe()
```

The output presents count, min, max, mean, standard deviation and I, II and III quartile information for each column.

	A	B
count	4.000000	4.000000
mean	59.435000	70.410000
std	27.259261	14.569059
min	30.420000	54.320000
25%	41.355000	60.830000
50%	57.660000	70.160000
75%	75.740000	79.740000
max	92.000000	87.000000

If the DataFrame contains character data, then *describe( )* method represents different statistics. Consider a new DataFrame **df_char** that consist of some character values in two columns.

```
df_char = pd.DataFrame({'One':['a', np.nan, 'b', 'a', 'd', np.nan], 'Two':[np.nan, 'd', 'c', np.nan, 'a', 'a']})
df_char
```

	One	Two
0	a	NaN
1	NaN	d
2	b	c
3	a	NaN
4	d	a
5	NaN	a

On calling the describe( ) method for **df_char** DataFrame:

```
df_char.describe()
```

Output will presents count of values, number of unique values, top value and frequency of top value.

	One	Two
count	4	4
unique	3	3
top	a	a
freq	2	2

Computing Correlation between DataFrame Values

Correlation is an important concept to study in data analysis. It helps in finding out relationship between two variables. The correlation value is in between -1 and +1. When correlation value is close to +1, it indicates

that two variables are positively correlated with each other. But if the correlation value is close to -1, it indicates that two variables are negatively correlated with each other. Pandas provide *corr( )* method to compute correlation between values of two columns.

To understand the concept of correlation, consider a DataFrame **df_stud** that consist of student name, attendance and marks for 5 students.

```
df_stud = pd.DataFrame({'Name':['Anu', 'Rima', 'Neha', 'Priya', 'Neetu'],
                        'Attendance':[87,84,98,33,65],
                        'Marks':[43, 40, 49, 12, 30]})
df_stud
```

	Name	Attendance	Marks
0	Anu	87	43
1	Rima	84	40
2	Neha	98	49
3	Priya	33	12
4	Neetu	65	30

Suppose we want to find the relationship between attendance and marks that whether high attendance has positive impact on marks or not. For that *corr( )* method can be used.

Calculate the correlation between the columns ‘Attendance’ and ‘Marks’

```
df_stud['Attendance'].corr(df_stud['Marks'])
```

And output is:

```
0.9994207200920612
```

Output represents that attendance and marks are positively related with each other, if the student is regular in attending classes then s/he has good chances of scoring good marks.

READING/WRITING DATA FROM/TO EXCEL OR CSV FILE

Generally, Data analyst work with very large volume of data stored in the form of a table i.e., having rows and columns. This large volume of data is stored in the form of file in the computer system so that it can be accessed whenever required. The process of data analysis may require fetching data from various sources. It can be fetching data from a file stored in the computer system, accessing it from some website and accessing it from the database. Data in the files can be stored in any format. Pandas support reading data from various file formats such as

csv, excel, JSON. This section is about how data can be read from or written into excel or csv file.

For reading and writing data from these file formats, Pandas provide built in methods. These methods read the data from specific file and convert it into a DataFrame. Similarly the data of the DataFrame can be stored in the specified file format.

Reading Data from Excel/CSV File

To read data from excel file, Pandas provide *read_excel( )* method. Consider an excel file named **'Data1'** that consist of two sheets with the names **'Employee'** and **'Student'**. The **'Employee'** sheet stores data about name, age and occupation of few employees while **'Student'** sheet stores data about name, subject and marks of few students.

The following statement will read the content from excel file **'Data1'** and convert it into a DataFrame:

```
df_ex = pd.read_excel("Data1.xlsx")
df_ex
```

While calling the *read_excel( )*, the mandatory argument is the name of the file along with its extension. The file Data1 is stored in the same folder where the python file is stored that is why only file name is given. However, if the excel file is stored in some other folder, then complete file path need to be mentioned with file name.

On executing the above statements, the output will be:

	Name	Age	Occupation
0	Anil	32	Teacher
1	Suresh	25	Data Analyst
2	Sujata	44	Scientist
3	Anne	24	Manager
4	Rima	30	Supervisor
5	Hema	33	Professor

Although the excel file contains data in two sheets, by default *read_excel( )* method reads data from the first sheet. To read data from a specific sheet, the argument **sheet_name** need to be used.

```
df_ex = pd.read_excel("Data1.xlsx", sheet_name="Student")
df_ex
```

Once you specify the name of the specific sheet using **sheet_name** argument, that specific sheet will be read by *read_excel( )* method and

then converted into the DataFrame.

The output of the above statements will be:

	Name	Subject	Marks
0	Supriya	Maths	98
1	Dion	English	93
2	Firoz	Physics	87
3	Rakesh	Chemistry	88
4	Meera	Physics	91
5	Sonia	Maths	95

Similar to *read_excel( )* method, Pandas provide *read_csv( )* method to read data from CSV (Comma Separated value) file. Consider a csv file **'Data2'** that stores information about employees.

The following statement will read the content from csv file **'Data2'** and convert it into a DataFrame:

```
df_cs = pd.read_csv("Data2.csv")
df_cs
```

The output of the above statement will be:

	Name	Age	Occupation
0	Anil	32	Teacher
1	Suresh	25	Data Analyst
2	Sujata	44	Scientist
3	Anne	24	Manager
4	Rima	30	Supervisor
5	Hema	33	Professor

Writing Data into Excel/CSV File

Suppose after reading data from the file and converting it into a DataFrame, you have made certain changes in the data. To save these changes in the excel/csv file, Pandas provide *to_excel( )* and *to_csv( )* methods, for excel and csv file, respectively.

Consider the following lists **'salary'** and **'address'**.

```
salary = [70000, 90000, 95000, 78000, 45000, 97000]
```

```
address = ['Delhi', 'Calcutta', 'Delhi', 'Bombay', 'Pune', 'Delhi']
```

Reading the sheet **'Student'** from excel file **'Data1'** and convert it into

DataFrame **df_stud**.

```
df_stud = pd.read_excel("Data1.xlsx", sheet_name="Student")
df_stud
```

The output will be:

	Name	Subject	Marks
0	Supriya	Maths	98
1	Dion	English	93
2	Firoz	Physics	87
3	Rakesh	Chemistry	88
4	Meera	Physics	91
5	Sonia	Maths	95

Now, add a new column **'Address'** in the DataFrame **df_stud**.

```
df_stud['Address']= address
df_stud
```

The output will be:

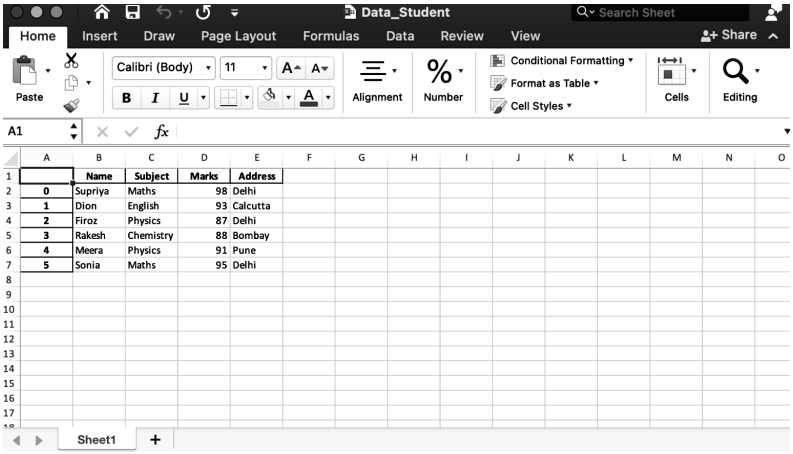
	Name	Subject	Marks	Address
0	Supriya	Maths	98	Delhi
1	Dion	English	93	Calcutta
2	Firoz	Physics	87	Delhi
3	Rakesh	Chemistry	88	Bombay
4	Meera	Physics	91	Pune
5	Sonia	Maths	95	Delhi

To update this information in excel file also, execute the following statement:

```
df_stud.to_excel("Data_Student.xlsx")
```

The mandatory argument with *to_excel( )* method is the file name with extension (along with file path if you want to store the file in separate folder).

The above statement will create an excel file with the updated information.



Similarly, information can be updated in the csv file. Consider the DataFrame df_cs that reads csv file 'Data2'.

```
df_cs = pd.read_csv("Data2.csv")
df_cs
```

The output will be:

	Name	Age	Occupation
0	Anil	32	Teacher
1	Suresh	25	Data Analyst
2	Sujata	44	Scientist
3	Anne	24	Manager
4	Rima	30	Supervisor
5	Hema	33	Professor

Now, add a new column **'Salary'** in the DataFrame **df_cs**.

```
df_cs['Salary'] = salary
df_cs
```

The output will be:

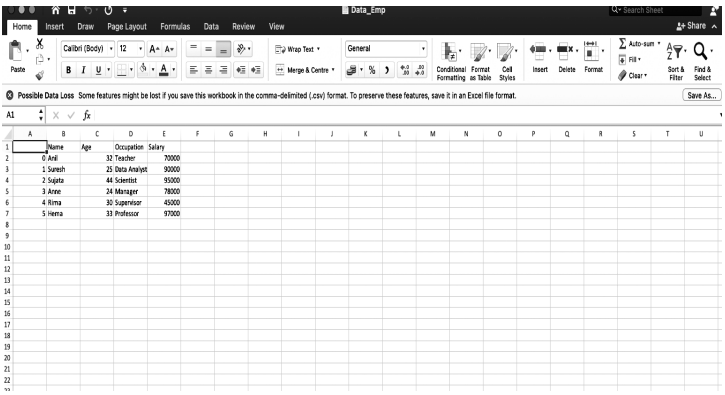
	Name	Age	Occupation	Salary
0	Anil	32	Teacher	70000
1	Suresh	25	Data Analyst	90000
2	Sujata	44	Scientist	95000
3	Anne	24	Manager	78000
4	Rima	30	Supervisor	45000
5	Hema	33	Professor	97000

To update this information in csv file also, execute the following statement:

```
df_cs.to_csv("Data_Emp.csv")
```

The mandatory argument with *to_csv( )* method is the file name with extension (along with file path if you want to store the file in separate folder).

The above statement will create a csv file with the updated information.



HANDLING MISSING DATA

Most of the time in Data Analysis is invested in preparing the data which includes cleaning, transforming and rearranging data. Cleaning data involves handling missing data in the dataset. It is important to do analysis on missing data also, to identify data collection problems or to check biases in the data due to missing data. Numeric missing data is generally represented by NaN (Not a Number) value.

Filtering Missing Data

Pandas provide *dropna( )* method to remove NaN values from the DataFrame. When used with Pandas series, *dropna( )* method remove all NaN values from the series. But when used with DataFrame, *dropna( )* method by default remove all the rows that contains one or more NaN value. However, it is possible to remove only those rows where all values are NaN or to remove rows by setting a threshold value.

Consider the following series **ser_miss** created from **list1**:

```
list1 = [70, np.nan, 85.2, 57, np.nan, 30]

ser_miss = pd.Series(list1)
ser_miss
```

The series will look like:

```
0    70.0
1     NaN
2    85.2
3    57.0
4     NaN
5    30.0
```

To remove NaN values from the series:

```
ser_miss.dropna()
```

The output will be:

```
0    70.0
2    85.2
3    57.0
5    30.0
```

Now, consider the following DataFrame **df_miss**, created from nested list **nest_list**:

```
nest_list = [[23, 45.5, np.nan, 65], [np.nan, 48.5, np.nan, 76],
             [np.nan, np.nan, np.nan, np.nan], [35, 23, 76, 45]]
```

```
df_miss = pd.DataFrame(nest_list)
df_miss
```

The output will be a DataFrame having 4 rows and 4 columns:

	0	1	2	3
0	23.0	45.5	NaN	65.0
1	NaN	48.5	NaN	76.0
2	NaN	NaN	NaN	NaN
3	35.0	23.0	76.0	45.0

On calling dropna( ) method for DataFrame df_miss:

```
df_clean = df_miss.dropna()
df_clean
```

It will remove all the rows having one or more NaN values and output will be:

	0	1	2	3
3	35.0	23.0	76.0	45.0

To remove only those rows, where all the values are NaN, use **how** argument:

```
df_clean = df_miss.dropna(how='all')
df_clean
```

And the output will be:

	0	1	2	3
0	23.0	45.5	NaN	65.0
1	NaN	48.5	NaN	76.0
3	35.0	23.0	76.0	45.0

It is also possible to delete the columns having NaN values by using the **axis** argument. To do this, let's add a new column in DataFrame **df_miss**. The column will contain all NaN values.

```
df_miss[4] = np.nan
df_miss
```

Now, **df_miss** will look like:

	0	1	2	3	4
0	23.0	45.5	NaN	65.0	NaN
1	NaN	48.5	NaN	76.0	NaN
2	NaN	NaN	NaN	NaN	NaN
3	35.0	23.0	76.0	45.0	NaN

To drop the last column from DataFrame:

```
df_clean = df_miss.dropna(axis=1, how='all')
df_clean
```

And the output will be:

	0	1	2	3
0	23.0	45.5	NaN	65.0
1	NaN	48.5	NaN	76.0
2	NaN	NaN	NaN	NaN
3	35.0	23.0	76.0	45.0

As discussed above, rows having NaN values can be removed from the DataFrame by setting a threshold value. For that thresh argument is used.

Following statement will remove only those rows from **df_miss** that have 3 or more NaN values:

```
df_clean = df_miss.dropna(thresh=3)
df_clean
```

The output will be:

	0	1	2	3	4
0	23.0	45.5	NaN	65.0	NaN
3	35.0	23.0	76.0	45.0	NaN

Filling Missing Data

Instead of removing the missing value, a good approach is to fill the missing values by using some appropriate value. It is required in the scenario when the dataset contains a large amount of missing values and removing them all will reduce the size of the dataset.

Pandas provide *fillna( )* method to fill the missing values. These values can be filled by using a constant value for all missing values or by using a distinct value for each column's missing value. A good approach is to fill the missing values by using the mean or median of dataset values.

Consider the **df_miss** DataFrame

	0	1	2	3	4
0	23.0	45.5	NaN	65.0	NaN
1	NaN	48.5	NaN	76.0	NaN
2	NaN	NaN	NaN	NaN	NaN
3	35.0	23.0	76.0	45.0	NaN

To fill the NaN values with a constant value **'10'**:

```
df_fill = df_miss.fillna(10)
df_fill
```

The output will be:

	0	1	2	3	4
0	23.0	45.5	10.0	65.0	10.0
1	10.0	48.5	10.0	76.0	10.0
2	10.0	10.0	10.0	10.0	10.0
3	35.0	23.0	76.0	45.0	10.0

To fill each column's NaN values with a distinct value, a dictionary can be passed as an argument to specify which column's NaN value will be filled by which value.

```
df_fill = df_miss.fillna({0:10, 1:20, 2:30, 3:40, 4:50})
df_fill
```

The output will be:

	0	1	2	3	4
0	23.0	45.5	30.0	65.0	50.0
1	10.0	48.5	30.0	76.0	50.0
2	10.0	20.0	30.0	40.0	50.0
3	35.0	23.0	76.0	45.0	50.0

To fill NaN values using mean value of the DataFrame:

```
df_fill = df_miss.fillna(df_miss.mean())
df_fill
```

The output will be:

	0	1	2	3	4
0	23.0	45.5	76.0	65.0	NaN
1	29.0	48.5	76.0	76.0	NaN
2	29.0	39.0	76.0	62.0	NaN
3	35.0	23.0	76.0	45.0	NaN

To fill NaN values and saving the change in the DataFrame, inplace argument need to be used. This will permanently save the filled values in the DataFrame.

```
df_miss.fillna({0:10, 1:20, 2:30, 3:40, 4:50}, inplace=True)
df_miss
```

The output will be:

	0	1	2	3	4
0	23.0	45.5	30.0	65.0	50.0
1	10.0	48.5	30.0	76.0	50.0
2	10.0	20.0	30.0	40.0	50.0
3	35.0	23.0	76.0	45.0	50.0

DATA TRANSFORMATION

Data transformation involves changing data in the form required for analysis. It includes removing duplicate values, replacing values with the specific values, changing form of data using mapping and dividing data into groups called bins.

Removing Duplicates

To remove duplicate value, Pandas provide *drop_duplicates( )* method.

Before removing the duplicate values, it is possible to check whether duplicate values exist or not. This can be done by calling *duplicated( )* method. Both the methods by default works for complete DataFrame i.e., duplicity of value is checked for each row (combining values of all columns). If you wish to check whether duplicate value exist in a particular column or not, column name need to be specified as the argument of the method.

Consider the DataFrame **df_duplicate** created from dictionary **dict1**. The DataFrame contains information about name and marks of students.

```
dict1 = {'Name':['Anuja', 'Kirti', 'Alen', 'Ramnik', 'Kirti', 'Alen'],
         'Marks':[89, 75, 94, 82, 74, 95]}

df_duplicate = pd.DataFrame(dict1)
df_duplicate
```

	Name	Marks
0	Anuja	89
1	Kirti	75
2	Alen	94
3	Ramnik	82
4	Kirti	74
5	Alen	95

Calling the duplicated( ) method to check whether duplicate value exist in the DataFrame or not.

```
df_duplicate.duplicated()
```

The duplicated( ) method returns a series of Boolean values indicating True if the duplicate value exist, otherwise False.

```
0    False
1    False
2    False
3    False
4    False
5    False
```

As there is no duplicate row in **df_duplicate** DataFrame, therefore, the series in the output contains all values as False.

To check the duplicate value for the column **'Name'**:

```
df_duplicate.duplicated(['Name'])
```

The output will be the series indicating last two values as duplicate.

```
0    False
1    False
2    False
3    False
4     True
5     True
```

Calling *drop_duplicates( )* method to drop duplicate values from DataFrame **df_duplicate**:

```
df_duplicate.drop_duplicates()
```

The output will contain the unique rows only.

	Name	Marks
0	Anuja	89
1	Kirti	75
2	Alen	94
3	Ramnik	82
4	Kirti	74
5	Alen	95

Calling *drop_duplicates( )* method to drop duplicate values from **'Name'** column of DataFrame **df_duplicate:**

```
df_duplicate.drop_duplicates(['Name'])
```

The output will be:

	Name	Marks
0	Anuja	89
1	Kirti	75
2	Alen	94
3	Ramnik	82

By default, *drop_duplicates( )* method keeps the first occurrence of value. In order to keep the last occurrence of value and remove the first occurrence, use the **keep** argument.

```
df_duplicate.drop_duplicates(['Name'], keep='last')
```

The output will be:

	Name	Marks
0	Anuja	89
3	Ramnik	82
4	Kirti	74
5	Alen	95

Transforming Data using Mapping

In certain data analysis operations, it is required to add the column and the values of the columns are mapped according to the existing column values of the dataset. For example, If a dataset consist of data about name, subject and marks of the students and a new column indicating that whether student is a science or humanities student needs to be added in the dataset then *map( )* method of Pandas can be used. A pre-defined mapping is passed as an argument to the **map( )** method.

Consider a DataFrame **df_student** created from the dictionary dict1:

```
dict1 = {'name': ['Vinay', 'Kushal', 'Aman', 'Saif', 'Anny', 'Riya', 'Pihu'],
        'subject': ['Maths', 'English', 'Chemistry', 'Physics', 'Computer', 'GK', 'Economics'],
        'marks': [22, 25, 24, 28, 23, 20, 27]}
```

```
df_student = pd.DataFrame(dict1)
df_student
```

The output will be:

	name	subject	marks
0	Vinay	Maths	22
1	Kushal	English	25
2	Aman	Chemistry	24
3	Saif	Physics	28
4	Anny	Computer	23
5	Riya	GK	20
6	Pihu	Economics	27

Consider the following dictionary **'discipline_mapping'** that defines the mapping between subject and their displine:

```
discipline_mapping = {'Maths':'Science', 'Physics':'Science', 'Chemistry':'Science',
                      'Computer':'Science', 'GK':'Humanities', 'English':'Humanities',
                      'Economics':'Humanities'}
```

To map the values of **subject** column of DataFrame **df_student** with a new column values **Discipline**, map( ) method can be used as:

```
df_student['Discipline'] = df_student['subject'].map(discipline_mapping)
```

The above statement will add a new column **Discipline** in the DataFrame **df_student** and add the values in the column as per the mapping defined in dictionary **discipline_mappping** given as argument.

```
df_student
```

The output will be:

	name	subject	marks	Discipline
0	Vinay	Maths	22	Science
1	Kushal	English	25	Humanities
2	Aman	Chemistry	24	Science
3	Saif	Physics	28	Science
4	Anny	Computer	23	Science
5	Riya	GK	20	Humanities
6	Pihu	Economics	27	Humanities

Replacing Values

We have already discussed the *fillna( )* method to fill the missing values. This method is very useful for handling missing or NaN values, but if the process of analysis requires replacing some mistyped values, then *fillna ( )* method cannot be used.

For this Pandas provide *replace( )* method that can replace single or multiple values in the dataset. Multiples values can be replaced by same value or by different values.

Consider the DataFrame **df_student** taken in previous section:

	name	subject	marks	Discipline
0	Vinay	Maths	22	Science
1	Kushal	English	25	Humanities
2	Aman	Chemistry	24	Science
3	Saif	Physics	28	Science
4	Anny	Computer	23	Science
5	Riya	GK	20	Humanities
6	Pihu	Economics	27	Humanities

Suppose you want to replace the discipline **'Humanities'** with **'Arts'**. For that *replace( )* method can be called as:

```
df_student.replace('Humanities', 'Arts')
```

The output will be:

	name	subject	marks	Discipline
0	Vinay	Maths	22	Science
1	Kushal	English	25	Arts
2	Aman	Chemistry	24	Science
3	Saif	Physics	28	Science
4	Anny	Computer	23	Science
5	Riya	GK	20	Arts
6	Pihu	Economics	27	Arts

Suppose you want to round off the marks of the students such that those who have scored less than 25, there marks will be replaced with 20.

To replace multiple values, *replace( )* method can be called as:

```
df_student.replace([21, 22, 23, 24], 20)
```

Multiple values can be passed as a list in the argument and the output will be:

	name	subject	marks	Discipline
0	Vinay	Maths	20	Science
1	Kushal	English	25	Humanities
2	Aman	Chemistry	20	Science
3	Saif	Physics	28	Science
4	Anny	Computer	20	Science
5	Riya	GK	20	Humanities
6	Pihu	Economics	27	Humanities

Suppose you want to replace all the values less than 23 with 20 and values equal to or greater than 23 with 25. For such scenario where different replacement is required for different values, *replace( )* method can be called as:

```
df_student.replace([21, 22, 23, 24], [20, 20, 25, 25])
```

The output will be:

	name	subject	marks	Discipline
0	Vinay	Maths	20	Science
1	Kushal	English	25	Humanities
2	Aman	Chemistry	25	Science
3	Saif	Physics	28	Science
4	Anny	Computer	25	Science
5	Riya	GK	20	Humanities
6	Pihu	Economics	27	Humanities

Binning the Data

Binning operation is required to group the continuous data. Consider a dataset that contain marks of the students. These marks can be grouped by creating the bin that can help in analysing that how many students have scored marks in a certain range. Pandas provides *cut( )* method that creates the bins of data passed to it as argument.

Consider the same DataFrame **df_student** that stores name, subject, marks and discipline of the students. Let's assume the marks are given out of 30.

	name	subject	marks	Discipline
0	Vinay	Maths	22	Science
1	Kushal	English	25	Humanities
2	Aman	Chemistry	24	Science
3	Saif	Physics	28	Science
4	Anny	Computer	23	Science
5	Riya	GK	20	Humanities
6	Pihu	Economics	27	Humanities

Let's divide the marks into bins of 15 to 20, 21 to 25 and 26 to 30. And call the *cut( )* method for binning the values of **marks** column of **df_student** DataFrame:

```
bins = [15, 20, 25, 30]

category = pd.cut(df_student['marks'], bins)

category
```

The output will be:

```
0     (20, 25]
1     (20, 25]
2     (20, 25]
3     (25, 30]
4     (20, 25]
5     (15, 20]
6     (25, 30]
```

Here, parentheses means that the value is excluded and square brackets means that the value is included.

If you want, you can count the values falling in a particular bin, by using *value_counts( )* method:

```
pd.value_counts(category)
```

The output will be:

```
(20, 25]    4
(25, 30]    2
(15, 20]    1
```

To change that which value will be included and which will be excluded, **right** argument need to be used.

```
bins = [15, 20, 25, 30]

category = pd.cut(df_student['marks'], bins, right=False)

category
```

The output will be:

```
0    [20, 25)
1    [25, 30)
2    [20, 25)
3    [25, 30)
4    [20, 25)
5    [20, 25)
6    [25, 30)
```

This will change the count of values falling a specific bins.

```
pd.value_counts(category)
```

```
[20, 25)    4
[25, 30)    3
[15, 20)    0
```

MERGING AND CONCATENATING DATA

Sometimes, the data is spread into multiple data frames and to perform analysis, it is required to merge the data from different data frames. Pandas provide *merge( )* and *concat( )* methods to merge and concatenate the data frames, respectively.

merge( ) method merge multiple data frames using common columns/keys and *concat( )* method concatenate data frames.

Merging Data Frames

To understand the *merge( )* method in detail, consider the sales data of a retail store spread across multiple files, namely, **market_fact**, **cust_dimen**, **prod_dimen**, **shipping_dimen** and **orders_dimen**.

Read the data from these files into DataFrames **market_df**, **customer_df**, **product_df**, **shipping_df** and **orders_df**.

```
market_df = pd.read_csv("D:/global_sales_data/market_fact.csv")
customer_df = pd.read_csv("D:/global_sales_data/cust_dimen.csv")
product_df = pd.read_csv("D:/global_sales_data/prod_dimen.csv")
shipping_df = pd.read_csv("D:/global_sales_data/shipping_dimen.csv")
orders_df = pd.read_csv("D:/global_sales_data/orders_dimen.csv")
```

Displaying first five records of the DataFrame market_df:

```
market_df.head()
```

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	0.93	0.54
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	14.30	0.37
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	26.30	0.38

The **market_df** contains the sales data of each order. While

customer_df, **product_df**, **shipping_df** and **orders_df** contains metadata about customers, products, shipping details and order details.

Displaying first five records of the DataFrame customer_df:

```
customer_df.head()
```

	Customer_Name	Province	Region	Customer_Segment	Cust_id
0	MUHAMMED MACINTYRE	NUNAVUT	NUNAVUT	SMALL BUSINESS	Cust_1
1	BARRY FRENCH	NUNAVUT	NUNAVUT	CONSUMER	Cust_2
2	CLAY ROZENDAL	NUNAVUT	NUNAVUT	CORPORATE	Cust_3
3	CARLOS SOLTERO	NUNAVUT	NUNAVUT	CONSUMER	Cust_4
4	CARL JACKSON	NUNAVUT	NUNAVUT	CORPORATE	Cust_5

Displaying first five records of the DataFrame product_df:

```
product_df.head()
```

	Product_Category	Product_Sub_Category	Prod_id
0	OFFICE SUPPLIES	STORAGE & ORGANIZATION	Prod_1
1	OFFICE SUPPLIES	APPLIANCES	Prod_2
2	OFFICE SUPPLIES	BINDERS AND BINDER ACCESSORIES	Prod_3
3	TECHNOLOGY	TELEPHONES AND COMMUNICATION	Prod_4
4	FURNITURE	OFFICE FURNISHINGS	Prod_5

Displaying first five records of the DataFrame shipping_df:

```
shipping_df.head()
```

	Order_ID	Ship_Mode	Ship_Date	Ship_id
0	3	REGULAR AIR	20-10-2010	SHP_1
1	293	DELIVERY TRUCK	02-10-2012	SHP_2
2	293	REGULAR AIR	03-10-2012	SHP_3
3	483	REGULAR AIR	12-07-2011	SHP_4
4	515	REGULAR AIR	30-08-2010	SHP_5

Displaying first five records of the DataFrame orders_df:

```
orders_df.head()
```

	Order_ID	Order_Date	Order_Priority	Ord_id
0	3	13-10-2010	LOW	Ord_1
1	293	01-10-2012	HIGH	Ord_2
2	483	10-07-2011	HIGH	Ord_3
3	515	28-08-2010	NOT SPECIFIED	Ord_4
4	613	17-06-2011	HIGH	Ord_5

Suppose you want to select all orders and observe Sales of the customer segment Corporate. Since customer segment details are present in the DataFrame **customer_df,** so it will be required to merge **customer_df** with **market_df**.

This merging can be done as:

```
df_1 = pd.merge(market_df, customer_df, how='inner', on='Cust_id')
df_1.head()
```

Merging operation is similar to applying joins in databases. Merging is of several types, including inner, outer, left and right. The type of merging is specified using ***how*** argument. As specified previously, merging is done using a common column between the DataFrames. This common column is specified using ***on*** argument.

The output of the above calling of *merge( )* method will be:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin	Customer_Name	Province	Region
Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56	AARON BERGMAN	ALBERTA	WEST
Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	0.93	0.54	AARON BERGMAN	ALBERTA	WEST
Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59	AARON BERGMAN	ALBERTA	WEST
Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	14.30	0.37	AARON BERGMAN	ALBERTA	WEST
Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	26.30	0.38	AARON BERGMAN	ALBERTA	WEST

Now the orders made by the customers from Corporate segment can be subset as:

```
df_1.loc[df_1['Customer_Segment'] == 'CORPORATE', :]
```

And the output will be:

Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin	Customer_Name	Province	Region	Customer_Segment
Cust_1818	136.81	0.01	23	-30.51	3.60	0.56	AARON BERGMAN	ALBERTA	WEST	CORPORATE
Cust_1818	42.27	0.01	13	4.56	0.93	0.54	AARON BERGMAN	ALBERTA	WEST	CORPORATE
Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59	AARON BERGMAN	ALBERTA	WEST	CORPORATE
Cust_1818	2337.89	0.09	43	729.34	14.30	0.37	AARON BERGMAN	ALBERTA	WEST	CORPORATE
Cust_1818	4233.15	0.08	35	1219.87	26.30	0.38	AARON BERGMAN	ALBERTA	WEST	CORPORATE
...	...	...	...	...	...	...	...	...	...	...
Cust_637	611.16	0.04	46	100.22	4.98	0.40	YANA SORENSEN	NEWFOUNDLAND	ATLANTIC	CORPORATE
Cust_851	121.87	0.07	39	11.32	1.35	0.40	YANA SORENSEN	QUEBEC	QUEBEC	CORPORATE
Cust_851	41.06	0.04	4	-16.39	6.28	0.35	YANA SORENSEN	QUEBEC	QUEBEC	CORPORATE
Cust_1519	994.04	0.03	10	-335.06	35.00	NaN	YANA SORENSEN	YUKON	YUKON	CORPORATE
Cust_1519	159.41	0.00	44	34.68	0.98	0.52	YANA SORENSEN	YUKON	YUKON	CORPORATE

Concatenating Data Frames

Concatenation of data frames is easier than merging. Concatenation is performed when you have data frames having the same number of columns and you want to append them (pile one on the top of other) or having the same number of rows and you want to append them side by side.

Consider the following two DataFrames **df1** and **df2** having same number of columns:

```
df1 = pd.DataFrame({'Name': ['Aman', 'Joy', 'Rashmi', 'Saif'],
                    'Age': ['34', '31', '22', '33'],
                    'Gender': ['M', 'M', 'F', 'M']}
                   )

df2 = pd.DataFrame({'Name': ['Akhil', 'Asha', 'Preeti'],
                    'Age': ['31', '22', '23'],
                    'Gender': ['M', 'F', 'F']}
                   )
```

```
df1
```

	Name	Age	Gender
0	Aman	34	M
1	Joy	31	M
2	Rashmi	22	F
3	Saif	33	M

```
df2
```

	Name	Age	Gender
0	Akhil	31	M
1	Asha	22	F
2	Preeti	23	F

To concatenate the DataFrames **df1** and **df2**, one on the top of other, *concat( )* method can be called as:

```
pd.concat([df1, df2], axis = 0)
```

Or

```
df1.append(df2)
```

Both will produce the same result:

	Name	Age	Gender
0	Aman	34	M
1	Joy	31	M
2	Rashmi	22	F
3	Saif	33	M
0	Akhil	31	M
1	Asha	22	F
2	Preeti	23	F

To conacatenate DataFrames having same number of rows, consider following DataFrames **df1** and **df2**:

```
df1 = pd.DataFrame({'Name': ['Aman', 'Joy', 'Rashmi', 'Saif'],
                    'Age': ['34', '31', '22', '33'],
                    'Gender': ['M', 'M', 'F', 'M']}
                   )
df1
```

	Name	Age	Gender
0	Aman	34	M
1	Joy	31	M
2	Rashmi	22	F
3	Saif	33	M

```
df2 = pd.DataFrame({'School': ['RK Public', 'JSP', 'Carmel Convent', 'St. Paul'],
                    'Graduation Marks': ['84', '89', '76', '91']}
                   )
df2
```

	School	Graduation Marks
0	RK Public	84
1	JSP	89
2	Carmel Convent	76
3	St. Paul	91

Calling *cancat( )* method:

```
pd.concat([df1, df2], axis = 1)
```

The output will be:

	Name	Age	Gender	School	Graduation Marks
0	Aman	34	M	RK Public	84
1	Joy	31	M	JSP	89
2	Rashmi	22	F	Carmel Convent	76
3	Saif	33	M	St. Paul	91

SELF-ASSESSMENT QUESTIONS

Q1 Create a Series of employees that stores salary of 10 employees. Set the name of the employees as the index label in the Series.

Q2 Create a DataFrame that stores information about 5 library books. The DataFrame should contain Book Name, Price and Publisher.

Add a new column ‘Author Name’ to the DataFrame and fill

the column with appropriate distinct values.

Q3 Remove the second row of the DataFrame created in question 2. After removing second row, remove the column 'Publisher' from the DataFrame.

Q4 Create a DataFrame that stores 10 numbers and their cubes in separate columns. Use lambda expression to multiply each cube value with 5 and display the result.

Q5 Sort the DataFrame created in question 2 in descending order of book name. Also set the name of the author as index in the DataFrame.

Q6 Create a DataFrame that store marks of 5 different subjects (Physics, Mathematics, Chemistry, English and Economics) for 10 students. The subject names should be the column name and the student id should be the index label.

Calculate the descriptive statistics for the DataFrame created. Also, compute the correlation between marks scored in Physics and Mathematics to find relationship between them.

Q7 Create one Excel file containing details about the weather (date, temperature) of two cities (Delhi and Bombay) in two different sheets. The weather details should be of 10 different dates.

Read the data from sheet that has details about weather of Bombay and add a column 'Rainy' to it. The value of the column should be either Yes or No.

Update the information in Excel file also.

Q8 Create a DataFrame containing information about age of 5 family members from 10 different families. The family member number should be the column name and the family number should be the index labels. Fill some values as NaN.

Perform the following operations on the DataFrame:

(i) Remove the rows having more than two NaN values.

(ii) Replace all NaN values with a constant value '25'.

(iii) Replace NaN values of each column with some distinct value.

(iv) Replace all NaN values with the mean value of the DataFrame.

Q9 Check if there is any duplicate value in the DataFrame created in question 8 and if exist, remove these duplicate values.

Q10 Create bins of the ages of column 1 from the DataFrame created in question 8 and count the number of values in each bin.

UNIT – 5
DATA VISUALIZATION (MATPLOTLIB)

INTRODUCTION

Data cleansing and normalization are essential processes in data analytics that help in improving the quality and consistency of data. Data Cleansing, also known as data cleaning or data scrubbing, refers to the process of identifying and correcting or removing errors, inconsistencies and inaccuracies from a dataset. The goal of the data cleansing is to ensure that the data is accurate, complete and reliable for analysis.

Some common tasks involved in data cleansing include:

- *Removing duplicate records:* Identifying and eliminating duplicate entries within a dataset to avoid skewing analysis results.
- *Handling missing values:* Dealing with missing or null values by either imputing them using statistical techniques or removing the incomplete records.
- *Correcting inconsistencies:* Resolving discrepancies and inconsistencies in data by standardizing formats, resolving conflicts and ensuring data integrity.
- *Validating data:* Verifying the accuracy of data by checking for outliers, logical errors or data that falls outside expected range.
- *Formatting data:* Converting data into a consistent format, such as converting dates to a standard format or ensuring consistent units of measurement.

By performing data cleansing, data analyst can enhance the quality of data and minimize the potential impact of errors on subsequent analysis or modelling.

PYTHON LIBRARIES FOR DATA VISUALIZATION

Python offers several powerful libraries for data visualization that provide a wide range of options to create informative and visually appealing charts, graphs and plots.

Some popular Python libraries for data visualization are:

- *Matplotlib***:** Matplotlib is one of the most widely used libraries for creating static, animated and interactive visualizations in Python. It provides a MATLAB like interface and supports various types of plots, including line plots, scatter plots, bar plots, histograms, pie charts and more. Matplotlib offers extensive customization options and is highly customizable.
- *Seaborn:* Seaborn is a statistical data visualization library built on the top of Matplotlib. It provides a higher-level interface and offers a range of aesthetic enhancements and statistical functionalities. Seaborn simplifies the creation of complex statistical visualizations, such as heatmaps, violin plots, box plots and joint distribution plots.
- *Plotly:* Plotly is a versatile library for creating interactive and dynamic visualizations. It supports a wide range of chart types, including line plots, scatter plots, bar charts, 3D plots and maps. Plotly allows to create interactive visualizations that can be embedded in web applications or shared online. It also offers a Python API, as well as interfaces for other programming languages.
- *Bokeh:* Bokeh is a library for interactive visualizations that targets modern web browsers. It enables the creation of interactive plots, dashboards and data applications. Bokeh supports a variety of plot types, including line plots, scatter plots, bar plots and heatmaps. It provides flexible interactivity options and can handle large datasets efficiently.
- *ggplot:* ggplot is a Python implementation of the popular ggplot2 library in R. It follows the grammar of graphics approach and allows users to create visually appealing and good-quality visualizations. ggplot simplifies the creation of layered plots and offers a wide range of customization options.
- *Altair:* Altair is a declarative statistical visualization library that leverages the Vega-Lite grammar. It provides a concise and intuitive syntax for creating a wide variety of visualizations, including scatter plots, line charts, bar charts and more. Altair supports interactive features and allows easy specifications of data transformations and visual encodings.

These libraries provide a rich set of tools for data visualization in Python. Depending on the specific requirements, dataset characteristics and desired output, the most suitable library can be chosen to create informative and visually appealing visualizations for the data analysis tasks.

WORKING WITH MATPLOTLIB

Matplotlib is a python library used for data visualization. Matplotlib contains the ***pyplot*** module, one of the most commonly used module for creating a variety of plots such as line plots, bar plots, histograms, scatter plot, pie chart and so on. To create these plots, ***pyplot*** contains a collection of functions such as plot( ), title( ), show( ) etc. It also provides the facility of creating multiple plots in the same figure. Pyplot also supports working with images by reading the image from file and looking at the attributes of images such as shape and data type.

TYPES OF CHARTS/GRAPHS

Matplotlib can create various types of plots and visualizations. It provides a wide range of options for customizing and enhancing the appearance of plots. Different types of plots that can be created using matplotlib are:

1. *Line Plot:* It is the most basic type of plot. It displays data points connected via straight lines. It is generally used to represent the trend or progression of data over time.
2. *Scatter Plot:* This plot displays information or individual data points as markers in two-dimensional plane. It is useful for displaying or visualizing the relationship between two variables. With the help of scatter plots, Data Analyst identify the patterns and clusters in the data.
3. *Bar Plot:* These plots are useful in representing categorical data. They represent data as rectangular bars where the length of each bar corresponds to the value of the data. Bar plots are generally used for comparing different categories or groups.
4. *Pie Chart:* These are circular plots divided into sections, where each section represents a category or group. The area of the section corresponds to the proportion or percentage of data belonging to that category or group.
5. *Histogram:* These plots are generally used to visualize the distribution of numerical data. Histograms divide the range of data into bins (intervals) and display the frequency or count of data points falling into each interval.
6. *Box Plot:* This plot is also known as box and whisker plot. It displays the distribution of numerical data through quartiles. Box plot also provide information about the median, range and potential outliers in the data.

7. *Heatmap*: Heatmaps use intensity of colors to represent values in a two-dimensional array. It is generally used to visualize the correlations, relationship between variables.

CREATING LINE PLOT

As discussed in the previous section, Line plot is generally used to represent the trend or progression of data over time. To create any type of plot using matplotlib, first step is to import the required library.

This statement will import the pyplot module of the matplotlib library with the alias name **'plt'**.

```
import matplotlib.pyplot as plt
```

Now, consider the following data to create points in the plot:

```
x1=[1,3,5,7,9]
y1=[23,45,67,80,90]

x2=[2,4,6,8,10]
y2=[67,70,82,55,92]
```

To plot these points *plot( )* method of *pyplot* module is used as:

```
plt.plot(x1,y1, label='Line1')
plt.plot(x2,y2, label='Line2')
```

label here is used to label the multiple lines in the graph. If the graph contains single line, then label is not required.

To display the labels in the graph after assigning names to them using **label** argument of *plot( )* method, it is mandatory to execute the *legend( )* method as:

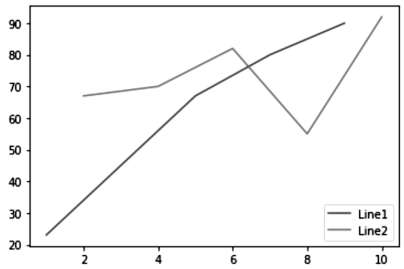
```
plt.legend(loc=4)
```

The **loc** argument in the *legend( )* method defines the position where the labels will be displayed inside graph. The permissible values of loc are 1, 2, 3 and 4.

To display the plot, *show( )* method is executed as:

```
plt.show()
```

The output of the above statements will be the following line plot:



CREATING SCATTER PLOT

Scatter plots are useful for displaying or visualizing the relationship between two variables. To create the scatter plot, consider the following data points:

```
x1=[1,3,5,7,9,11,13,15,17,19,21,23]
y1=[23,45,67,80,90,33,45,56,22,78,88,67]

x2=[2,4,6,8,10]
y2=[67,70,82,55,92]
```

To plot these data points, *scatter( )* method of *pyplot* module is used as:

```
plt.scatter(x1,y1, c='red', s=20, alpha=0.4, marker='^')
plt.scatter(x2,y2, c='violet', s=40, marker= '*')
```

The mandatory arguments here are the data point variables. Other arguments specified in the statement are used for the following purpose:

The argument **'c'** specifies the color of the marker, while **'s'** specifies the size of the marker. Another argument **'alpha'** specifies the gradient of the color of the marker. The range of alpha is from 0 to 1. The **'marker'** argument specifies the shape of the marker. Permissible values for the marker are ^ (for triangle), **o** (for circle, the default value), **s** (for square), ***** (for star shape).

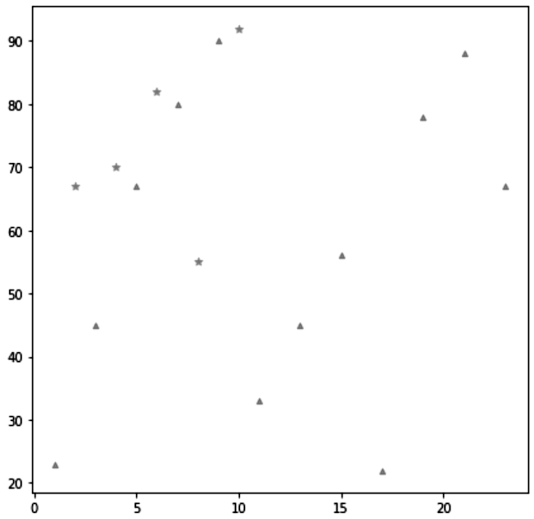
Apart from setting these arguments, the size of the figure or plot can also be controlled by using *figure( )* method. As argument, this method accepts the height and width of the figure.

```
plt.figure(figsize=(7,7))
```

To display the plot:

```
plt.show()
```

The output of the above statements will be the following scatter plot:



CREATING BAR GRAPH

Bar graph/plots are useful in representing categorical data. Consider the following data points:

```
X =[10,20,30,40,50,60]
Y= [20,45,67,55,11,23]
```

To plot these data points, *bar( )* method of *pyplot* module is used as:

```
plt.bar(X,Y, color = "red", width=3)
```

The **'color'** argument is used to specify the color of the bars and **'width'** argument is used to specify that how wide the bars will be in the graph.

The *title( )* method is used to specify a title for the graph.

```
plt.title("Bar Graph")
```

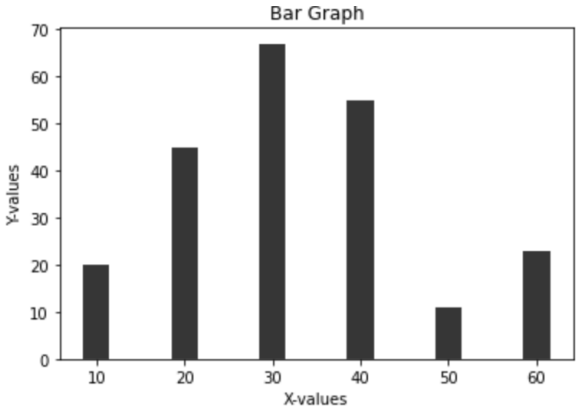
The *xlabel( )* and *ylabel( )* methods are used to specify the labels for x-axis and y-axis.

```
plt.xlabel("X-values")
plt.ylabel("Y-values")
```

To display the plot:

```
plt.show()
```

The output of the above statements will be the following bar plot:



CREATING PIE CHART

Pie charts are circular plots divided into sections, where each section represents a category or group. Consider the following lists **'x'**, **'activities'** and **'cols'**.

List **'x'** represents the data i.e., number of persons who have performed some activities, **'activities'** represents the names of activities performed and **'cols'** represents the color for each section.

```
x=[7,2,2,13,5,15]
activities=['sleeping', 'eating', 'working', 'playing', 'music', 'study']
cols=['c', 'g', 'r', 'b', 'pink', 'yellow']
```

To set the size of the plot:

```
plt.figure(figsize=(15,8))
```

To plot these data points, *pie( )* method of *pyplot* module is used as:

```
plt.pie(x,labels=activities, colors=cols, startangle=0, explode=(0.1,0.1,0,0,0,0), autopct='%1.2f%%')
```

Here, **labels** specify the labels of the sections of the chart, **colors** specify the color for each section, **startangle** specify the position angle to start sections, **explode** to slice out the section (a value of 0.1 for first two activities means these two sections will be sliced out a bit), **autopct** specify that the percentage of graph share will be calculated automatically up to two decimal points.

To specify the title of the plot:

```
plt.title("Interesting Graph\ncheck it out")
```

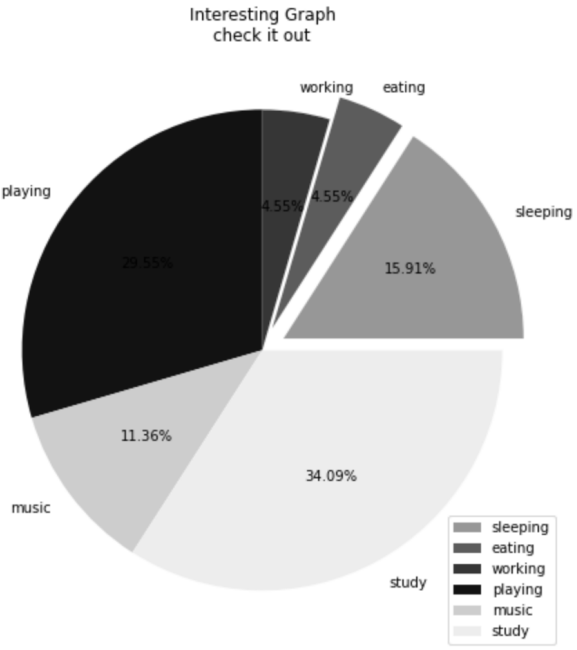
legend( ) method to display the labels in the chart with location value equals to 4.

```
plt.legend(loc=4)
```

To display the chart:

```
plt.show()
```

The output of the above statements will be the following pie chart:



CREATING HISTOGRAM

Histograms are generally used to visualize the distribution of numerical data. Histograms divide the range of data into bins (intervals) and display the frequency or count of data points falling into each interval.

Consider the following data points:

```
x=[7,2,2,13,5,15]
```

To plot these data points, *hist( )* method of *pyplot* module is used as:

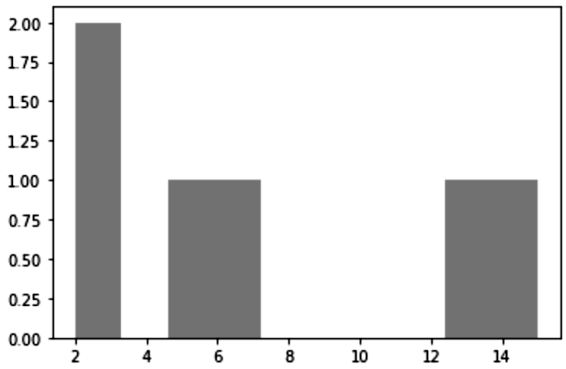
```
plt.hist(x, color='violet')
```

Color argument here specify the color of the histogram plot.

To display the histogram:

```
plt.show()
```

The output of the above statements will be the following histogram:



Combining two different Types of Graphs/Plots

It is also possible to combine two different types of plots. Consider the following data points:

```
x1=[1,3,5,7,9]
y1=[23,45,67,80,90]
```

The same data points can be used to create two different types of plots in the same x-axis and y-axis. Let's create bar and line plot together using above data points.

Using *bar( )* and *plot( )* methods to plot the bar and line plot:

```
plt.bar(x1,y1, label="Bar Graph", color="yellow")
plt.plot(x1,y1, label="Line Plot", color="red")
```

Specifying plot title:

```
plt.title("Bar-Line Graph")
```

Specifying plot labels:

```
plt.xlabel("X")
plt.ylabel("Y")
```

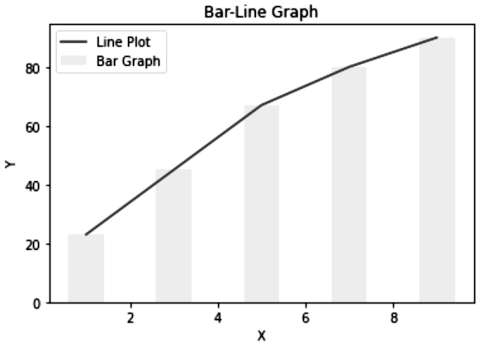
Using *legend( )* method to display labels:

```
plt.legend()
```

To display the plots:

```
plt.show()
```

The output of the above statements will be the following bar-line plot:



CREATING FIGURES AND SUBPLOTS

Sometimes it is required to create multiple plots in the same figure. Matplotlib supports the concept of figures and subplots that can be used to create multiple subplots inside the same figure.

To create multiple plots in the same figure, *subplot( )* method of *pyplot* module is used. The syntax of the *subplot( )* method is as follows:

plt.subplot(nrows, ncols, nsubplots)

where *nrows* specifies number of rows, *ncols* specifies number of columns and *nsubplots* specifies number of the sub-plot in the row.

Consider an example of creating a figure having 4 subplots:

Initializing data points using *linspace( )* method of NumPy library:

```
x = np.linspace(1, 10, 100)
```

Do not forget to import NumPy library before executing the above statement.

For subplot 1:

```
plt.subplot(2, 2, 1)
plt.title("Linear")
plt.plot(x, x)
```

For subplot 2:

```
plt.subplot(2, 2, 2)
plt.title("Cubic")
plt.plot(x, x**3)
```

For subplot 3:

```
plt.subplot(2, 2, 1)
plt.title("Log")
plt.plot(x, np.log(x))
```

For subplot 4:

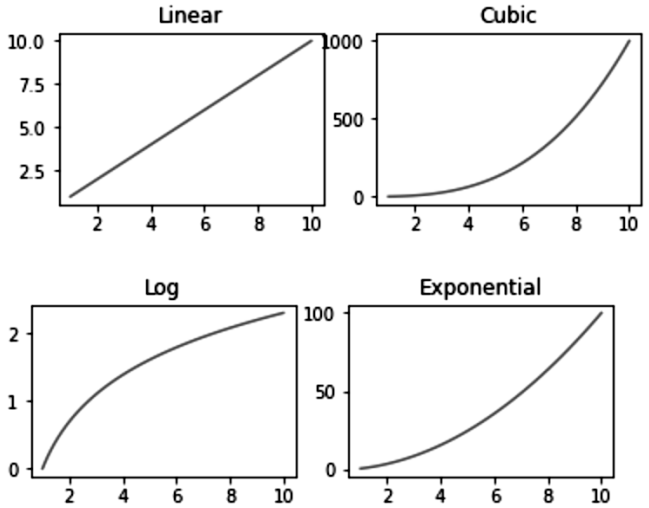
```
plt.subplot(2, 2, 2)
plt.title("Exponential")
plt.plot(x, x**2)
```

In each subplot, *subplot( )* method is creating the figure by defining number of rows, number of column and the number of the subplot in the row. The *title( )* method defines the title for the plots and *plot( )* method draws the plot.

To display the plot:

```
plt.show()
```

The output of the above statements will be the following figure having four subplots:



CREATING BOX PLOT

Also known as box and whisker plot, it displays the distribution of numerical data through quartiles. Box plot also provides information about the median, range and potential outliers in the data.

To draw the box plot, consider the global sales data stored in **'market_fact'** csv file.

Reading data from csv file (Do not forget to import Pandas library before executing this statement):

```
df = pd.read_csv("D:/global_sales_data/market_fact.csv")
df.head()
```

The output will be:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	0.93	0.54
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	14.30	0.37
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	26.30	0.38

Let's draw the box plot of **'Order_Quantity'** column:

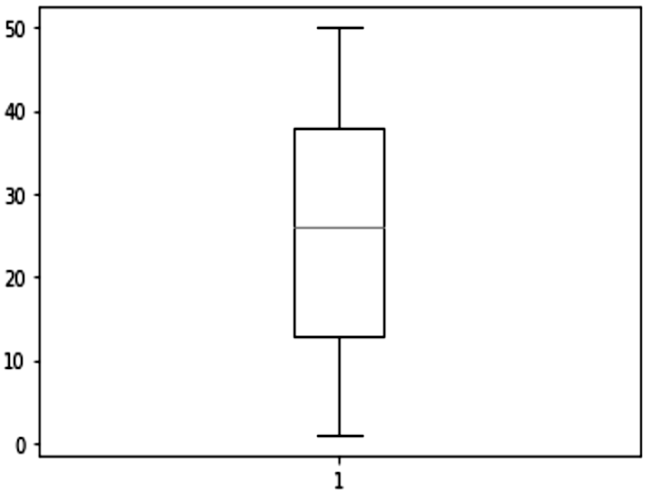
To plot the column, *boxplot( )* method of *pyplot* module is used as:

```
plt.boxplot(df['Order_Quantity'])
```

To show the box plot:

```
plt.show()
```

The output of the above statements will be the following box plot:



Here, the centre orange line represents the median value. The two horizontal lines at the bottom and top represents the minimum and maximum values and the outliers are presented above or below the horizontal lines.

WORKING WITH IMAGES

Matplotlib can also read images using *imread( )* method. Internally, it

reads and stores images as an array. The array can be used for various data manipulation tasks, just like a normal array.

Reading the image using *imread( )* method:

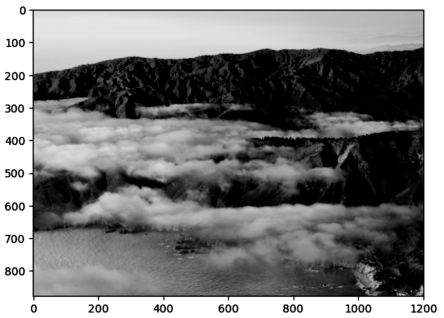
```
image = plt.imread("Example Image.png")
```

To display the image:

```
plt.imshow(image)

plt.show()
```

The output will be the image read from the system:



SELF-ASSESSMENT QUESTIONS

Q1 Consider the attendance and marks of 10 students for two different class sections. Here, attendance will be represented on x-axis and marks will be represented on y-axis, for each class section.

Perform the following operation:

(i) Draw and display the Line plot for both class sections in one graph.

(ii) Draw and display the Scatter plot for both class sections in one graph.

Q2 Consider the age of 10 persons represented as x-axis data points and duration (in number of hours) spend in watching T.V. as y-axis data points.

Draw and display the bar plot for these data points.

Q3 Consider the following data points that represents the number of employees in each department: X = [9, 7, 5, 10, 8, 15].

The name of the departments are given as: Dept = [‘HR’,

‘Material’, ‘Accounts’, ‘Finance’, ‘Testing’, ‘Development’].

Define the distinct colors for each department.

Draw and display the Pie chart to show the percentage of employees working in each department.

Q4 Consider the marks of 10 students ranging from 30 to 100. Draw and display the histogram for the data points.

Q5 Combine the Line plot and Scatter plot created in question 1. Draw and display the combined Line-Scatter plot.