Introduction to Jupyter/ IPython

- IPython was originally developed by Fernando Perez in 2001 as an enhanced Python interpreter.
- In 2014, Project Jupyter started as a spin-off project from IPython.
- Jupyter provides following packages
  - **Jupyter notebook:** A web-based interface to programming environments of Python.
  - **QtConsole**: Qt based terminal for Jupyter kernels similar to IPython.
  - **Nbviewer**: Facility to share Jupyter notebooks.
  - **JupyterLab**: Modern web based integrated interface for all products.

Working with Data

- Data science applications require data by definition.
- Data requires by the application are stored in a different location and different format.
- We need techniques required to access data in several forms and locations.
  - Memory form – it represents a form of data store that your computer support natively.
  - Flat Files – it represents data are store on a computer hard drive in file format.
  - Relation database – in this large data are stored on the internet and small-sized data may be stored in the small file that appears on your hard drive as well.
- Storing data in local computer memory represents the fastest and most reliable means to access it.
- The data could reside anywhere. However, we don't interact with the data in its storage location. We load the data into memory from the storage location and interact with it in memory.

Basic IO operations in Python

- When it comes to storing, reading, or communicating data, working with the files of an operating system is both necessary and easy with Python.
- Before we can read or write a file, we have to open it using Python's built-in open() function.

File Open

- The open function is used for opening files.

  Syntax :

```
fileobject = open(filename [, accessmode][, buffering])
```

  - **filename** – it specifies the name of the file we want to open.

- **accessmode** – it determines the mode in which the file has to be opened.
- **buffering -** If buffering is set to 0, no buffering will happen, if set to 1 line buffering will happen, if greater than 1, then buffering action is performed with the indicated buffer size and if negative is given it will follow the system default buffering behaviour.
- We can specify a relative path in argument to **open** method, alternatively, we can also specify an absolute path.
- To specify absolute path,
  - In windows, f=open('D:\\folder\\subfolder\\filename.txt')
  - In mac & linux, f=open('/user/folder/subfolder/filename.txt')

- We suppose to close the file once we are done using the file in the Python using **close()** method.

```
f = open('college.txt')
---
f.close()
```

- List of possible file modes are list below

Mode	Description
r	Read only (default)
rb	Read only in binary format
r+	Read and Write both
rb+	Read and Write both in binary format
w	Write only
wb	Write only in binary format
w+	Read and Write both
wb+	Read and Write both in binary format
a	Opens file to append, if file not exist will create it for write
ab	Append in binary format, if file not exist will create it for write
a+	Append, if file not exist it will create for read & write both
ab+	Read and Write both in binary format

Read file in Python

- For reading file content in python following functions are available.
  - **read() :**
    - read() function is used for reading the full contents of a file.
    - **read(size)** will read specified bytes from the file, if we don't specify size it will return the whole file.

      Code:

```
f = open('college.txt')
data = f.read()
print(data)
```

Output:

```
Darshan Institute of Engineering and Technology - Rajkot
At Hadala, Rajkot - Morbi Highway,
Guiarat-363650. INDIA
```

- **readlines() :**
  - It will return a list of lines from the file.

  Code:

```
f = open('college.txt')
lines = f.readlines()
print(lines)
```

  Output:

```
['Darshan Institute of Engineering and Technology - Rajkot\n', 'At Hadala,
Rajkot - Morbi Highway,\n', 'Gujarat-363650, INDIA']
```

- **Iterate files :**
  - We can use for loop to get each line separately,

  Code:

```
f = open('college.txt')
lines = f.readlines()
for l in lines :
    print(l)
```

  Output:

```
Darshan Institute of Engineering and Technology - Rajkot
At Hadala, Rajkot - Morbi Highway,
Gujarat-363650, INDIA
```

"with" statement

- It is used for error handling.
- We may have typo in the filename or file we specified is moved/deleted, in such cases, there will be an error while executing code.
- To handle such situations we can use the new syntax of opening the file using **with** keyword.
- It allows us to ensure that a resource is "cleaned up" when the code that uses it finishes running, even if exceptions are thrown.
- The with statement itself ensures proper acquisition and release of resources
- with statement in your objects will ensure that you never leave any resource open.

```
with open('college.txt') as f :
    data = f.read()
    print(data)
```

- When we open a file using with we need not to close the file.

Write file in Python

- write() method will write the specified data to the file.

```
with open('college.txt','a') as f :
    f.write('Hello world')
```

- If we open a file with '**w**' mode it will overwrite the data to the existing file or will create a new file if the file does not exist.
- If we open a file with '**a**' mode it will append the data at the end of the existing file or will create a new file if the file does not exist.

Reading CSV files without any library functions

- A comma-separated values file is a delimited text file that uses a comma to separate values.
- Each line is a data record, Each record consists of many fields, separated by commas.

Example (Book.csv):

```
studentname,enrollment,cpi
abcd,123456,8.5
bcde,456789,2.5
cdef,321654,7.6
```

- We can use Microsoft Excel to access CSV files.
- In the future, we will access CSV files using different libraries, but we can also access CSV files without any libraries.

```
with open('Book.csv') as f :
  rows = f.readlines()
  isFirstLine = True # to ignore column headers
  for r in rows :
    if isFirstLine :
        isFirstLine = False
        continue
    cols = r.split(',')
    print('Student Name = ', cols[0], end=" ")
    print('\tEn. No. = ', cols[1], end=" ")
    print('\tCPI = \t', cols[2])
```

Output:

```
Student Name =  abcd      En. No. =  123456   CPI =   8.5

Student Name =  bcde      En. No. =  456789   CPI =   2.5

Student Name =  cdef      En. No. =  321654   CPI =   7.6
```

NumPy

- NumPy stands for *Numerical Python*
- NumPy is a Python library used for working with arrays.
- It provides the function to work with linear algebra, Fourier transform, and matrices.
- It is an open-source project and we can use it freely.
- It is a library consisting of multidimensional array objects and a collection of routines for processing those arrays.
- Using NumPy, mathematical, and logical operations on arrays can be performed very easily.
- In Python, we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy provides an array object that is very much faster than traditional Python lists.
- The array object in NumPy is called **ndarray**.
- Arrays are very frequently used in data science, where speed and resources are very important.

  Install :
  - conda install numpy *or*
  - pip install numpy
- Once we installed NumPy, import it in your applications by adding the import keyword

```
import numpy as np
--
```

- NumPy is usually imported under the np alias. Alias is an alternate name for referring to the same thing. now NumPy package can be referred to as np instead of numpy.

NumPy Array

- The most important object defined in NumPy is an N-dimensional array type called **ndarray**.
- It describes the collection of items of the same type, Items in the collection can be accessed using a zero-based index
- An instance of ndarray class can be constructed in many different ways, the basic ndarray can be created as below.

  Syntax :

```
import numpy as np

numpy.array(object [, dtype = None][, copy = True][, order = None][,
subok = False][, ndmin = 0])
```

- The above constructor of array takes the following parameters :

Parameters	Description
object	Any object exposing the array interface method returns an array, or any (nested) sequence like list, tuple, set, and dict.
dtype	The desired data type of array, An optional parameter.
copy	By default (true), the object is copied, an optional parameter.
order	C (row major) or F (column major) or A (any) (default), optional parameter.
subok	By default, the returned array is forced to be a base class array. If true, sub-classes passed through.
ndmin	Specifies minimum dimensions of the resultant array.

Example:

```
import numpy as np
a= np.array(['darshan','Insitute','rajkot'])
print(type(a))
print(a)
```

Output:

```
<class 'numpy.ndarray'>
['darshan' 'Insitute' 'rajkot']
```

NumPy Array creation routines

- The NumPy array can be created from other low-level constructors of ndarray.

numpy.empty

- It creates an uninitialized array of specified shape and dtype

Syntax

```
numpy.empty(shape[, dtype = float][, order = 'C'])
```

- The above constructor takes the following parameters :

Parameters	Description
shape	Shape of an empty array in int or tuple of int
dtype	Desired data type of array, optional parameter
order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Example:

```
import numpy as np
x = np.empty([3,2], dtype = int)
print x
```

Output:

```
[[140587109587816 140587109587816]
 [140587123623488 140587124774352]
 [ 94569341940000  94569341939976]]
```

numpy.zero

- zeros(n) function will return NumPy array of a given shape, filled with zeros.

Syntax

```
numpy.zeros(shape[, dtype = float][, order = 'C'])
```

The above constructor takes the following parameters:

Parameters	Description
shape	Shape of an empty array in int or tuple of int
dtype	Desired data type of array, optional parameter
order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Example:

```
import numpy as np
c = np.zeros(3)
print(c)
c1 = np.zeros((3,3)) #have to give as tuple
print(c1)
```

Output:

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

numpy.ones

- ones(n) function will return NumPy array of a given shape, filled with ones.

Syntax

```
numpy.ones(shape[, dtype = float][, order = 'C'])
```

The above constructor takes the following parameters:

Parameters	Description
shape	Shape of an empty array in int or tuple of int
dtype	Desired data type of array, optional parameter
order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Example:

```
import numpy as np
c = np.ones(3)
print(c)
c1 = np.ones((3,3)) #have to give as tuple
print(c1)
```

Output:

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

numpy.arange

- it will create NumPy array starting from start till the end (not included) with specified steps.

Syntax

```
numpy.arange(start, stop[, step][, dtype])
```

The above constructor takes the following parameters:

Parameters	Description
start	The start of an interval. If omitted, defaults to 0
stop	The end of an interval (not including this number)
step	Spacing between values, default is 1
dtype	Desired data type of array, optional parameter

Example:

```
import numpy as np
b = np.arange(0,10,1)
print(b)
```

Output:

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

numpy.linspace

- linspace function will return evenly spaced numbers over a specified interval.

Syntax

```
numpy.linspace(start, stop, num, endpoint, retstep, dtype)
```

The above constructor takes the following parameters:

Parameters	Description
start	The starting value of the sequence
stop	The end value of the sequence, included in the sequence if endpoint set to true
num	The number of evenly spaced samples to be generated. Default is 50

endpoint	True by default, hence the stop value is included in the sequence. If false, it is not included
retstep	If true, returns samples and step between the consecutive numbers
dtype	Data type of output ndarray

Example:

```
import numpy as np
c = np.linspace(0,1,11)
print(c)
```

Output:

```
[0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.]
```

numpy.logspace

- logspace function will return evenly spaced numbers over a specified interval on a log scale. The scale indicates of base usually 10.

Syntax

```
numpy.logspace(start, stop, num, endpoint, base, dtype)
```

The above constructor takes the following parameters:

Parameters	Description
start	The starting value of the sequence
stop	The end value of the sequence, included in the sequence if endpoint set to true
num	The number of evenly spaced samples to be generated. Default is 50
endpoint	True by default, hence the stop value is included in the sequence. If false, it is not included
base	Base of log space, default is 10
dtype	Data type of output ndarray

Example:

```
import numpy as np
c = np.logspace(1.0,2.0, num = 10)
print(c)
```

Output:

```
[10.  12.91549665  16.68100537   21.5443469    27.82559402
   35.93813664   46.41588834   59.94842503   77.42636827  100.]
```

Attributes of NumPy

- this section describes attributes of NumPy

Attribute	Description	Example	Output

ndarray.shape	This array attribute returns a tuple consisting of array dimensions.	`import numpy as np`
		`a=np.array([[1,3],[4,6]])`
		`print a.shape`
			(2, 2)

ndarray.ndim	This array attribute returns the number of array dimensions.	`import numpy as np`
		`a = np.arange(24)`
		`print(a.ndim)`
			1

numpy.itemsize	This array attribute returns the length of each element of array in bytes.	`import numpy as np`
		`x = np.array([1,2,3,4,5])`
		`print x.itemsize`
			8

numpy.flags	This function returns the current values of a set of flags.	`import numpy as np`
		`x = np.array([1,2,3,4,5])`
		`print x.flags`
			`C_CONTIGUOUS : True`
			`F_CONTIGUOUS : True`
			`OWNDATA : True`
			`WRITEABLE : True`
			`ALIGNED : True`
			`UPDATEIFCOPY : False`

Random Numbers in NumPy

- It is used to generate a random number for the array.
- For generating random numbers NumPy offers the random module to work with random numbers.

```
from numpy import random
--
--
```

numpy.random.rand()

- Rand() function is used for generating random float numbers.

Syntax

```
rand(d0, d1, …, dn)
```

- Here d specifies dimensions.
- Rand function will create an n-dimensional array with random data using a uniform distribution, if we do not specify any parameter it will return a random float number.

Example:

```
import numpy as np
r1 = np.random.rand()
print(r1)
r2 = np.random.rand(3,2) # no tuple
print(r2)
```

Output:

```
0.23937253208490505

[[0.58924723 0.09677878] [0.97945337 0.76537675] [0.73097381 0.51277276]]
```

numpy.random.randint()

- It is used for generating random int numbers.
- randint(low, high, num) function will create a one-dimensional array with num random integer data between low and high.

```
randint(low[, high=None][, size=None][, dtype='l'])
```

- following are the parameters of function

Parameters	Description
low	Lowest integer number for random generation
high	Highest integer number for random generation
size	Total number of random number generation
dtype	Desired dtype of the array

Example:

```
import numpy as np
r3 = np.random.randint(1,100,10)
print(r3)
```

Output:

```
[78 78 17 98 19 26 81 67 23 24]
```

numpy.random.randn()

- Randn() function is used for generating random float numbers.

Syntax

```
randn(d0, d1, …, dn)
```

- Here d specifies dimensions.
- Randn() function will create an n-dimensional array with random data using the normal distribution, if we do not specify any parameter it will return a random float number.

Example:

```
import numpy as np
r1 = np.random.randn()
print(r1)
r2 = np.random.randn(3,2) # no tuple
print(r2)
```

Output:

```
-0.15359861758111037
[[ 0.40967905 -0.21974532]  [-0.90341482 -0.69779498] [ 0.99444948 -1.45308348]]
```

Multidimensional array in NumPy

- In Numpy, the number of dimensions of the array is called the rank of the array.
- A tuple of integers giving the size of the array along each dimension is known as shape of the array
- Elements in Numpy arrays are accessed by using square brackets and can be initialized by using nested Python Lists
- Now let's create 2d array.

Example:

```
arr = np.array([['a','b','c'],['d','e','f'],['g','h','i']])
print('double = ',arr[2][1]) # double bracket notaion
print('single = ',arr[2,1]) # single bracket notation
```

Output:

```
double = h
single = h
```

- There are two ways in which you can access element of multi-dimensional array, example of both the method is given below
- Both methods are valid and provide the same answer, but single bracket notation is recommended as in double bracket notation it will create a temporary sub-array of the third row and then fetch the second column from it.
- Single bracket notation will be easy to read and write while programming

Aggregations in NumPy

- NumPy provides many aggregate functions or statistical functions to work with a single-dimensional or multi-dimensional array.
- Following are the aggregation function of NumPy.

Function	Description
np.mean()	Compute the arithmetic mean along the specified axis.
np.std()	Compute the standard deviation along the specified axis.

np.var()	Compute the variance along the specified axis.
np.sum()	Sum of array elements over a given axis.
np.prod()	Return the product of array elements over a given axis.
np.cumsum()	Return the cumulative sum of the elements along a given axis.
np.cumprod()	Return the cumulative product of elements along a given axis.
np.min()	Return the minimum of an array or minimum along an axis.
np.max()	Return the maximum of an array or minimum along an axis.
np.argmin()	Returns the indices of the minimum values along an axis
np.argmax()	Returns the indices of the maximum values along an axis

- **Example**

```
l = [7,5,3,1,8,2,3,6,11,5,2,9,10,2,5,3,7,8,9,3,1,9,3]
a = np.array(l)
print('Min = ',a.min())
print('ArgMin = ',a.argmin())
print('Max = ',a.max())
print('ArgMax = ',a.argmax())
print('Sum = ',a.sum())
print('Mean = ',a.mean())
print('Std = ',a.std())
```

Output:

```
Min = 1
ArgMin = 3
Max = 11
ArgMax = 8
Sum = 122
Mean = 5.304347826086956
Std = 3.042235771223635
```

- When we apply aggregate functions with multidimensional ndarray, it will apply an aggregate function to all its dimensions (axis).

Example

```
import numpy as np
array2d = np.array([[1,2,3],[4,5,6],[7,8,9]])
print('sum = ',array2d.sum())
```

Output:

```
sum = 45
```

- If we want to get sum of rows or cols we can use the axis argument with the aggregate functions.

Example

```
import numpy as np
array2d = np.array([[1,2,3],[4,5,6],[7,8,9]])
print('sum (cols)= ',array2d.sum(axis=0)) #Vertical
print('sum (rows)= ',array2d.sum(axis=1)) #Horizontal
```

Output:

```
sum (cols) = [12 15 18]
sum (rows) = [6 15 24]
```

Indexing & Slicing in NumPy

- The Contents of ndarray object can be accessed and modified by indexing or slicing, just like Python's in-built container objects.
- Indexing means accessing any element from ndarray.
- Slicing in python means taking elements from one given index to another given index.
- Similar to Python List, we can use the same syntax to slice ndarray.

  Syntax:

  ```
  array[start:end:step]
  ```

  - Start – it represents the starting of slicing. Default value is 0
  - End – it represents the ending of slicing. Default value is length of array
  - Step – it represents the step of the slicing:
- Let's take one example of slicing a single dimensional array. Consider the following example for data.

  Example:

  ```
  import numpy as np
  arr = np.array(['a','b','c','d','e','f','g','h'])
  ```

Expression	Output
arr[::]	['a' 'b' 'c' 'd' 'e' 'f' 'g' 'h']
arr[::2]	['a' 'c' 'e' 'g']
arr[:7:]	['a' 'b' 'c' 'd' 'e' 'f' 'g']
arr[:7:2]	['a' 'c' 'e' 'g']
arr[2::]	['c' 'd' 'e' 'f' 'g' 'h']
arr[2::2]	['c' 'e' 'g']
arr[2:7:]	['c' 'd' 'e' 'f' 'g']
arr[2:7:2]	['c' 'e' 'g']
arr[2:5]	['c' 'd' 'e']
arr[:5]	['a' 'b' 'c' 'd' 'e']
arr[5:]	['f' 'g' 'h']
arr[::-1]	['h' 'g' 'f' 'e' 'd' 'c' 'b' 'a']

- Slicing a multi-dimensional array would be the same as a single dimensional array with the help of the single bracket notation we learn earlier.
- Let's take one example of slicing a multidimensional dimensional array. Consider the following example for data.

Example:

```
import numpy as np
arr = np.array([['a','b','c'],['d','e','f'],['g','h','i']])
```

Expression	Description	Output
arr[0:2 , 0:2]	first two rows and cols	[['a' 'b'] ['d' 'e']]
arr[::-1]	reversed rows	[['g' 'h' 'i'] ['d' 'e' 'f'] ['a' 'b' 'c']]
arr[: , ::-1]	reversed cols	[['c' 'b' 'a'] ['f' 'e' 'd'] ['i' 'h' 'g']]
arr[::-1,::-1]	complete reverse	[['i' 'h' 'g'] ['f' 'e' 'd'] ['c' 'b' 'a']]

- When we slice an array and apply some operation on them, it will also make changes in the original array, as it will not create a copy of an array while slicing.
- Array slicing is mutable.

Example:

```
import numpy as np
arr = np.array([1,2,3,4,5])
arrsliced = arr[0:3]

arrsliced[:] = 2 # Broadcasting

print('Original Array = ', arr)
```

Output:

```
Original Array = [2 2 2 4 5]
Sliced Array = [2 2 2]
```

NumPy Arithmetic Operators

- In NumPy array, we can perform a mathematical operation on every element with a single command.

Example:

```
import numpy as np
arr1 = np.array([[1,2,3],[1,2,3],[1,2,3]])
arr2 = np.array([[4,5,6],[4,5,6],[4,5,6]])
```

Expression	Description	Output
arr1 + 2	addition of matrix with scalar	[[3 4 5] [3 4 5] [3 4 5]]
arr1 + arr2	addition of two matrices	[[5 7 9] [5 7 9] [5 7 9]]
arr1 - 2	subtraction of matrix with scalar	[[-1 0 1] [-1 0 1] [-1 0 1]]
arr1 - arr2	subtraction of two matrices	[[-3 -3 -3] [-3 -3 -3] [-3 -3 -3]]
arr1 / 2	Division of matrix with *scalar*	[[0.5 1. 1.5] [0.5 1. 1.5] [0.5 1. 1.5]]
arr1 / arr2	Division of two matrix	[[0.25 0.4 0.5] [0.25 0.4 0.5] [0.25 0.4 0.5]]
arr1 * 2	multiply matrix with scalar	[[2 4 6] [2 4 6] [2 4 6]]
arr1 * arr2	multiply two matrices	[[4 10 18] [4 10 18] [4 10 18]]
np.matmul(arr1 ,arr2)	Matrix Multiplication	[[24 30 36] [24 30 36] [24 30 36]]
arr1.dot(arr2)	Dot	[[24 30 36] [24 30 36] [24 30 36]]

Sorting array in NumPy

- Sorting means putting elements in an ordered sequence.
- Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.
- The NumPy ndarray object has a function called sort(), which will sort a specified array.

- The sort method returns a copy of the array, leaving the original array unchanged.

Syntax:

```
import numpy as np
--
np.sort(arr,axis,kind,order)
```

- The above sort function takes the following parameters:

Parameters	Description
arr	array to sort (inplace)
axis	axis to sort (default=0)
kind	kind of algo to use, the default value is 'quicksort', we can also specify other algorithms like 'mergesort', 'heapsort'.
order	on which field we want to sort if multiple fields available

Example:

```
import numpy as np
arr = np.array(['Darshan','Rajkot','Insitute','of','Engineering'])
print("Before Sorting = ", arr)
arr.sort() # or np.sort(arr)
print("After Sorting = ",arr)
```

Output:

```
Before Sorting =  ['Darshan' 'Rajkot' 'Insitute' 'of' 'Engineering']
After Sorting =  ['Darshan' 'Engineering' 'Insitute' 'Rajkot' 'of']
```

- Sorting a multidimensional array using the sort function.

Example:

```
import numpy as np
dt = np.dtype([('name', 'S10'),('age', int)])
arr2 = np.array([('Darshan',200),('ABC',300),('XYZ',100)],dtype=dt)
arr2.sort(order='name')
print(arr2)
```

Output:

```
[(b'ABC', 300) (b'Darshan', 200) (b'XYZ', 100)]
```

Conditional Selection in NumPy

- Similar to arithmetic operations when we apply any comparison operator to Numpy Array, then it will be applied to each element in the array and a new bool Numpy Array will be created with values True or False.

Example:

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8,9,10])
print(arr)
boolArr = arr > 5
print(boolArr)
newArr = arr[arr > 5]
print(newArr)
```

Output:

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

- If we pass this bool Numpy Array to subscript operator [] of an original array then it will return a new Numpy Array containing elements from Original array for which there was True in bool Numpy Array.
- Here are some of the examples of conditional selections.

Description	Expression
divisible by 3	arr[arr%3==0]
even numbers	arr[arr%2==0]
Odd numbers	arr[arr%2!=0]
greater than 5 and less than 20	arr[(arr > 5) & (arr < 20)]

Pandas

- In 2008, developer Wes McKinney started developing pandas when in need of high performance, flexible tool for analysis of data.
- Before Pandas, Python was majorly used for data munging and preparation. It had very little contribution to data analysis. Pandas solved this problem.
- Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — load, prepare, manipulate, model, and analyze.
- Pandas is an open-source library built on top of NumPy.
- It allows for fast data cleaning, preparation, and analysis.
- It excels in performance and productivity.
- It also has built-in visualization features.
- It can work with data from a wide variety of sources.
- It has the following features.
    - Time Series functionality.
    - Fast and efficient DataFrame object with the default and customized indexing.
    - Tools for loading data into in-memory data objects from different file formats.
    - Data alignment and integrated handling of missing data.
    - Reshaping and pivoting of data sets.
    - Label-based slicing, indexing, and subsetting of large data sets.

- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High-performance merging and joining of data.

- **Install :**
  - conda install pandas *or*
  - pip install pandas
- Once we installed Pandas, import it in your applications by adding the import keyword

```
import pandas as pd
--
```

- Pandas is usually imported under the pd alias. Alias is an alternate name for referring to the same thing. now Pandas package can be referred to as pd instead of Pandas.

Series in Pandas

- Series is a one-dimensional* array with axis labels.
- It supports both integer and label-based index but the index must be of hashable type.
- If we do not specify an index it will assign an integer zero-based index.
- Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

Syntax:

```
import pandas as pd
s = pd.Series(data,index,dtype,copy=False)
```

- The above constructor takes the following parameters :

Parameters	Description
data	array like Iterable, dict, or scalar value
index	array-like or Index (1d)
dtype	The data type for the output Series. If not specified, this will be inferred from data. Optional argument
copy	Copy input data. Default is False

- Let's takes one example of pandas.

Example:

```
import pandas as pd
s = pd.Series([1, 3, 5, 7, 9, 11])
print(s)
```

Output:

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

- We can then access the elements inside Series just like the array using square brackets notation.
- We can specify the data type of Series using dtype parameter

Example:

```
import pandas as pd
s = pd.Series([1, 3, 5, 7, 9, 11], dtype='str')
print("S[0] = ", s[0])
b = s[0] + s[1]
print("Sum = ", b)
```

Output:

```
S[0] = 1
Sum = 13
```

- We can specify an index to Series with the help of the index parameter.

Example:

```
import numpy as np
import pandas as pd
i = ['name','address','phone','email','website']
d = ['darshan','rj',’123','d@d.com','darshan.ac.in']
s = pd.Series(data=d,index=i)
print(s)
```

Output:

```
name              darshan
address                rj
phone                 123
email             d@d.com
website    darshan.ac.in
dtype:object
```

Creating Time Series

- We can use some of the pandas inbuilt date functions to create a time series.

Example:

```
import numpy as np
import pandas as pd
dates = pd.to_datetime("27th of July, 2020")
i = dates + pd.to_timedelta(np.arange(5), unit='D')
d = [50,53,25,70,60]
time_series = pd.Series(data=d,index=i)
print(time_series)
```

Output:

```
2020-07-27    50
2020-07-28    53
2020-07-29    25
2020-07-30    70
2020-07-31    60
dtype: int64
```

Data Frames in Pandas

- Data frames are two-dimensional data structures, i.e. data is aligned in a tabular format in rows and columns.
- The Data frame also contains labeled axes on rows and columns.
- Features of Data Frame :
  - It is size-mutable.
  - Has labeled axes.
  - Columns can be of different data types.
  - We can perform arithmetic operations on rows and columns.

Structure :

	PDS	Algo	SE	INS
101				
102				
103				
....				
160				

- A pandas DataFrame can be created using the following constructor

Syntax:

```
import pandas as pd
df = pd.DataFrame(data,index,columns,dtype,copy=False)
```

- The above constructor takes the following parameters :

Parameters	Description
data	ndarray (structured or homogeneous), Iterable, dict, or DataFrame
index	array like row index
columns	array like col index
dtype	The data type for the output dataframe. If not specified, this will be inferred from data. Optional argument
copy	Copy data from inputs. Default False

- Let's take one example of pandas.

Example:

```
import numpy as np
import pandas as pd
randArr = np.random.randint(0,100,20).reshape(5,4)
df =
pd.DataFrame(randArr,np.arange(101,106,1),['PDS','Algo','SE','INS'])
print(df)
```

Output:

```
     PDS  Algo  SE  INS
101    0    23  93   46
102   85    47  31   12
103   35    34   6   89
104   66    83  70   50
105   65    88  87   87
```

- We can access columns, rows, and perform an operation on the data frame. Consider the below examples.

Action	Example	Output
Grabbing the column	`df['PDS']`	101 0 102 85 103 35 104 66 105 65 Name: PDS, dtype: int32
Grabbing the multiple column	`df['PDS', 'SE']`	PDS SE 101 0 93 102 85 31 103 35 6 104 66 70 105 65 87
Grabbing a row – for accessing row loc() function is used	`df.loc[101] Or` `df.iloc[0]`	PDS 0 Algo 23 SE 93 INS 46 Name: 101, dtype: int32

Grabbing Single Value	`df.loc[101, 'PDS']`	0
Deleting Row	`df.drop('103',inplace=True)`	PDS Algo SE INS 101 0 23 93 46 102 85 47 31 12 104 66 83 70 50 105 65 88 87 87
Creating new column and column Sum	`df['total'] = df['PDS'] + df['Algo'] + df['SE'] + df['INS']` `print(df)`	PDS Algo SE INS total 101 0 23 93 46 162 102 85 47 31 12 175 103 35 34 6 89 164 104 66 83 70 50 269 105 65 88 87 87 327
Deleting Column and Row	`df.drop('total',axis=1,inplace=True)`	PDS Algo SE INS 101 0 23 93 46 102 85 47 31 12 103 35 34 6 89 104 66 83 70 50 105 65 88 87 87
Getting Subset of Data Frame	`df.loc[[101,104], [['PDS','INS']]`	PDS INS 101 0 46 104 66 50
Selecting all cols except one	`df.loc[:, df.columns != 'Algo' ]`	PDS SE INS 101 0 93 46 102 85 31 12 103 35 6 89 104 66 70 50 105 65 87 87

Conditional Selection in Pandas

- Similar to NumPy we can do conditional selection in pandas.
- We can then use this boolean DataFrame to get associated values.
- It will set NaN (Not a Number) in case of False
- We can apply conditions to a specific column.

Example:

```
import numpy as np
import pandas as pd
np.random.seed(121)
randArr = np.random.randint(0,100,20).reshape(5,4)
df =
pd.DataFrame(randArr,np.arange(101,106,1),['PDS','Algo','SE','INS'])
print(df)
dfBool = df > 50
print(dfBool)
print(df[dfBool])
dfBool1 = df['PDS'] > 50
print(df[dfBool1])
```

Output:

```
       PDS   Algo     SE   INS
101     66     85      8    95
102     65     52     83    96
103     46     34     52    60
104     54      3     94    52
105     57     75     88    39

       PDS   Algo     SE    INS
101   True   True  False   True
102   True   True   True   True
103  False  False   True   True
104   True  False   True   True
105   True   True   True  False

       PDS   Algo     SE   INS
101     66     85    NaN    95
102     65     52     83    96
103    NaN    NaN     52    60
104     54    NaN     94    52
105     57     75     88   NaN

       PDS   Algo     SE   INS
101     66     85      8    95
102     65     52     83    96
104     54      3     94    52
105     57     75     88    39
```

Read CSV in Pandas

- read_csv() is used to read the Comma Separated Values (CSV) file into a pandas DataFrame.

Syntax:

```
pd.read_csv(filepath, sep, header, index_col)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
filePath	Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected
sep	Delimiter to use. (Default is comma)
header	Row number(s) to use as the column names.
index_col	index column(s) of the data frame.

Example:

```
dfINS = pd.read_csv('Marks.csv',index_col=0,header=0)
print(dfINS)
```

Output:

```
      PDS  Algo  SE   INS
101   50    55  60  55.0
102   70    80  61  66.0
103   55    89  70  77.0
104   58    96  85  88.0
201   77    96  63  66.0
```

Read Excel in Pandas

- Read an Excel file into a pandas DataFrame.
- Supports xls, xlsx, xlsm, xlsb, odf, ods and odt file extensions read from a local filesystem or URL. Supports an option to read a single sheet or a list of sheets.
- real_excel () function is used for reading excel file.

Syntax:

```
pd. read_excel(excelFile, sheet_name, header, index_col)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
excelFile	str, bytes, ExcelFile, xlrd.Book, path object, or file-like object
sheet_name	Sheet no in integer or the name of the sheet, can have a list of sheets.
index_col	Index column of the data frame.

Example:

```
df = pd.read_excel('records.xlsx', sheet_name='Employees')
print(df)
```

Output:

```
   EmpID    EmpName EmpRole
0      1     abc      CEO
1      2     xyz    Editor
2      3     pqr    Author
```

Read from MySQL Database

- We need two libraries for that
  - conda install sqlalchemy
  - conda install pymysql
- After installing both the libraries, import create_engine from sqlalchemy and import pymysql

```
from sqlalchemy import create_engine
import pymysql
```

- Then, create a database connection string and create engine using it.

```
db_connection_str = 'mysql+pymysql://username:password@host/dbname'
db_connection = create_engine(db_connection_str)
```

- After getting the engine, we can fire any sql query using pd.read_sql method.
- read_sql is a generic method that can be used to read from any sql (MySQL,MSSQL, Oracle etc…)

```
df = pd.read_sql('SELECT * FROM cities', con=db_connection)
print(df)
```

Output:

```
   CityID   CityName      City  Description    CityCode
0       1      Rajkot     Rajkot Description here       RJT
1       2  Ahemdabad  Ahemdabad Description here       ADI
2       3      Surat      Surat Description here       SRT
```

Setting/Resetting index in Pandas

- We have seen index does not have a name, if we want to specify a name to an index we can specify it using DataFrame.index.name property.

Syntax:

```
df.index.name('Index name')
```

- We can use pandas built-in methods to set or reset the index
- pd.set_index('NewColumn',inplace=True), will set new column as index,
- pd.reset_index(), will reset index to zero based numberic index.

Setting/Resetting index in Pandas

- Hierarchical indexes (AKA multi indexes) help us to organize, find, and aggregate information faster at almost no cost.
- Example where we need Hierarchical indexes

Numeric Index/Single Index:

```
      Col Dep  Sem   RN  S1  S2  S3
0       ABC  CE    5  101  50  60  70
1       ABC  CE    5  102  48  70  25
2       ABC  CE    7  101  58  59  51
3       ABC  ME    5  101  30  35  39
4       ABC  ME    5  102  50  90  48
5  Darshan  CE    5  101  88  99  77
6  Darshan  CE    5  102  99  84  76
7  Darshan  CE    7  101  88  77  99
8  Darshan  ME    5  101  44  88  99
```

Multi Index Index:

```
                RN  S1  S2  S3
Col     Dep Sem
ABC     CE  5    101  50  60  70
            5    102  48  70  25
            7    101  58  59  51
        ME  5    101  30  35  39
            5    102  50  90  48
Darshan CE  5    101  88  99  77
            5    102  99  84  76
            7    101  88  77  99
        ME  5    101  44  88  99
```

- Creating multi indexes is as simple as creating a single index using set_index method, only difference is in case of multi indexes we need to provide list of indexes instead of a single string index, let's see an example for that

Example:

```
dfMulti = pd.read_csv('MultiIndexDemo.csv')
dfMulti.set_index(['Col','Dep','Sem'],inplace=True)
print(dfMulti)
```

Output:

```
                RN  S1  S2  S3
Col     Dep Sem
ABC     CE  5    101  50  60  70
            5    102  48  70  25
            7    101  58  59  51
        ME  5    101  30  35  39
            5    102  50  90  48
Darshan CE  5    101  88  99  77
            5    102  99  84  76
            7    101  88  77  99
        ME  5    101  44  88  99
```

- Now we have a multi-indexed DataFrame from which we can access data using multiple indexes.

Example:

```
print(dfMulti.loc['Darshan']) # Sub DataFrame for all the students
of Darshan
print(dfMulti.loc['Darshan','CE']) # Sub DataFrame for Computer
Engineering students from Darshan
```

Output:

```
          RN  S1  S2  S3
Dep Sem
CE  5    101  88  99  77
    5    102  99  84  76
    7    101  88  77  99
ME  5    101  44  88  99

      RN  S1  S2  S3
Sem
5    101  88  99  77
5    102  99  84  76
7    101  88  77  99
```

Reading in Multiindexed DataFrame directly from CSV

- read_csv function of pandas provides an easy way to create multi-indexed DataFrame directly while fetching the CSV file.
- Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int/str is given, a MultiIndex is used.

Syntax:

```
pd.read_csv('MultiIndexDemo.csv',index_col=[{Comma separated column
index}])
```

Example:

```
dfMultiCSV = pd.read_csv('MultiIndexDemo.csv',index_col=[0,1,2])
print(dfMultiCSV)
```

Output:

```
                  RN  S1  S2  S3
Col     Dep Sem
ABC     CE  5    101  50  60  70
            5    102  48  70  25
            7    101  58  59  51
        ME  5    101  30  35  39
            5    102  50  90  48
Darshan CE  5    101  88  99  77
            5    102  99  84  76
            7    101  88  77  99
        ME  5    101  44  88  99
```

Cross Section in DataFrame

- The xs() function is used to get cross-section from the Series/DataFrame.

- This method takes a key argument to select data at a particular level of a MultiIndex.

Syntax:

```
DataFrame.xs(key, axis=0, level=None, drop_level=True)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
key	label
axis	Axis to retrieve cross-section
level	level of key
drop_level	False if you want to preserve the level

Example:

```
dfMultiCSV = pd.read_csv('MultiIndexDemo.csv', index_col=[0,1,2])
print(dfMultiCSV)
print(dfMultiCSV.xs('CE',axis=0,level='Dep'))
```

Output:

```
                  RN  S1  S2  S3
Col     Dep Sem
ABC     CE  5    101  50  60  70
            5    102  48  70  25
            7    101  58  59  51
        ME  5    101  30  35  39
            5    102  50  90  48
Darshan CE  5    101  88  99  77
            5    102  99  84  76
            7    101  88  77  99
        ME  5    101  44  88  99

RN  S1  S2  S3
Col     Sem
ABC     5    101  50  60  70
        5    102  48  70  25
        7    101  58  59  51
Darshan 5    101  88  99  77
        5    102  99  84  76
        7    101  88  77  99
```

Dealing with Missing Data

- There are many methods by which we can deal with the missing data, some of most commons are listed below,

dropna

- it will drop(delete) the missing data(rows/cols)

Syntax:

```
DataFrame.dropna(axis, how, inplace)
```

- The above method takes the following parameters :

Parameters	Description
axis	Determine if rows or columns which contain missing values are removed.
how	Determine if a row or column is removed from DataFrame when we have at least one NA or all NA. • 'any': If any NA values are present, drop that row or column. • 'all': If all values are NA, drop that row or column.
inplace	If True, do the operation in place and return None.

fillna

- It will fill specified values in place of missing data.

  Syntax:

```
DataFrame.interpolate(self, method='linear', axis=0, limit=None,
inplace=False, limit_direction='forward', limit_area=None,
downcast=None)
```

- The above method takes the following parameters :

Parameters	Description
value	Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.
method	Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use next valid observation to fill gap.
axis	Axis along which to fill missing values.
inplace	If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).
limit	If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill.
downcast	A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type

Interpolate

- it will interpolate missing data and fill interpolated values in place of missing data.

  Syntax:

```
DataFrame.interpolate(method='linear', axis=0, limit=None,
inplace=False, limit_direction='forward', limit_area=None,
downcast=None)
```

- The above method takes the following parameters :

Parameters	Description
method	Interpolation technique to use. One of:

	Linear, time, index, pad, nearest, zero, slinear, quadratic, cubic, spine, barycentric, krogh, from_derivatives.
axis	Axis to interpolate along.
inplace	Update the data in place if possible.
limit	The maximum number of consecutive NaNs to fill. Must be greater than 0.
limit_direction	If the limit is specified, consecutive NaNs will be filled in this direction.
limit_area	If the limit is specified, consecutive NaNs will be filled with this restriction.
downcast	Downcast dtypes if possible.

Groupby in Pandas

- Any groupby operation involves one of the following operations on the original object. They are
  - Splitting the Object
  - Applying a function
  - Combining the results
- In many situations, we split the data into sets and we apply some functionality on each subset.
- we can perform the following operations
  - Aggregation – computing a summary statistic
  - Transformation – perform some group-specific operation
  - Filtration – discarding the data with some condition
- Basic ways to use of groupby method
  - df.groupby('key')
  - df.groupby(['key1','key2'])
  - df.groupby(key,axis=1)
- Consider IPL database Example

Example:

```
import pandas as pd
ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings', 'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)
print df.groupby('Team').groups
groupIPL = df.groupby('Year')
for name,group in groupIPL :
  print(name)
  print(group)
```

Output:

```
{'Kings': Int64Index([4, 6, 7], dtype='int64'),
 'Devils': Int64Index([2, 3], dtype='int64'),
 'Riders': Int64Index([0, 1, 8, 11], dtype='int64'),
 'Royals': Int64Index([9, 10], dtype='int64'),
 'kings': Int64Index([5], dtype='int64')}
2014
   Points  Rank    Team  Year
0     876     1  Riders  2014
2     863     2  Devils  2014
4     741     3   Kings  2014
9     701     4  Royals  2014
2015
    Points  Rank    Team  Year
1      789     2  Riders  2015
3      673     3  Devils  2015
5      812     4   kings  2015
10     804     1  Royals  2015
2016
   Points  Rank    Team  Year
6     756     1   Kings  2016
8     694     2  Riders  2016
2017
    Points  Rank    Team  Year
7      788     1   Kings  2017
11     690     2  Riders  2017
```

Concatenation in Pandas

- Concatenation basically glues together DataFrames.
- Keep in mind that dimensions should match along the axis you are concatenating on.
- You can use pd.concat and pass in a list of DataFrames to concatenate together:
- We can use the axis=1 parameter to concatenate columns.

Example:

```
dfCX = pd.read_csv('CX_Marks.csv',index_col=0)
dfCY = pd.read_csv('CY_Marks.csv',index_col=0)
dfCZ = pd.read_csv('CZ_Marks.csv',index_col=0)
dfAllStudent = pd.concat([dfCX,dfCY,dfCZ])
print(dfAllStudent)
```

Output:

```
      PDS   Algo SE
101    50    55  60
102    70    80  61
103    55    89  70
104    58    96  85
201    77    96  63
202    44    78  32
203    55    85  21
204    69    66  54
301    11    75  88
302    22    48  77
303    33    59  68
304    44    55  62
```

Join in Pandas

- join() method will efficiently join multiple DataFrame objects by index(or column specified).

Syntax:

```
df.join(dfOther, on, header, how)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
dfOther	Right Data Frame
on	specify the column on which we want to join (Default is index)
how	• left - use calling frame's index (Default). • right - use dfOther index. • outer - form union of calling frame's index with other's index (or column if on is specified), and sort it. lexicographically. • inner - form intersection of calling frame's index (or column if on is specified) with other's index, preserving the order of the calling's one.

Example:

```
dfINS = pd.read_csv('INS_Marks.csv',index_col=0)
dfLeftJoin = allStudent.join(dfINS)
print(dfLeftJoin)
```

Output:

```
      PDS  Algo  SE   INS
101   50    55  60  55.0
102   70    80  61  66.0
103   55    89  70  77.0
104   58    96  85  88.0
201   77    96  63  66.0
202   44    78  32   NaN
203   55    85  21  78.0
204   69    66  54  85.0
301   11    75  88  11.0
302   22    48  77  22.0
303   33    59  68  33.0
304   44    55  62  44.0
```

Merge in in Pandas

- Merge DataFrame or named Series objects with a database-style join.
- Similar to the join method, but used when we want to join/merge with the columns instead of index.

Syntax:

```
object.merge(dfOther, on, left_on, right_on, how)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
dfOther	Right Data Frame
on	specify the column on which we want to join (Default is index)
left_on	specify the column of left Dataframe
right_on	specify the column of right Dataframe
how	• left - use calling frame's index (Default). • right - use dfOther index. • outer - form union of calling frame's index with other's index (or column if on is specified), and sort it. lexicographically. • inner - form intersection of calling frame's index (or column if on is specified) with other's index, preserving the order of the calling's one.

Example:

```
m1 = pd.read_csv('Merge1.csv')
print(m1)
m2 = pd.read_csv('Merge2.csv')
print(m2)
m3 = m1.merge(m2,on='EnNo')
print(m3)
```

Output:

```
   RollNo      EnNo Name
0     101  11112222  Abc
1     102  11113333  Xyz
2     103  22224444  Def

       EnNo  PDS  INS
0  11112222   50   60
1  11113333   60   70

   RollNo      EnNo Name  PDS  INS
0     101  11112222  Abc   50   60
1     102  11113333  Xyz   60   70
```

NumPy v/s Pandas

NumPy	Pandas
NumPy module works with numerical data.	Pandas module works with the tabular data.
NumPy has a powerful tool like Arrays.	Pandas has powerful tools like Series, DataFrame etc.
NumPy consumes less memory as compared to Pandas.	Pandas consume large memory as compared to NumPy.
NumPy provides a multi-dimensional array.	Pandas provide 2d table object called DataFrame.
Indexing of numpy Arrays is very fast.	Indexing of the pandas series is very slow as compared to numpy arrays.
if we want ease of coding we should use pandas.	if we want performance we should use NumPy,

Web Scrapping using Beautiful Soup

- Beautiful Soup is a library that makes it easy to scrape information from web pages.
- It sits atop an HTML or XML parser, providing Pythonic idioms for iterating, searching, and modifying the parse tree.

Example:

```
import requests
import bs4
req = requests.get('https://www.darshan.ac.in/DIET/CE/Faculty')
soup = bs4.BeautifulSoup(req.text,'lxml')
allFaculty = soup.select('body > main > section:nth-child(5) > div >
div > div.col-lg-8.col-xl-9 > div > div')
for fac in allFaculty :
    allSpans = fac.select('h2>a')
    print(allSpans[0].text.strip())
```

Output:

```
Dr. Gopi Sanghani
Dr. Nilesh Gambhava
Dr. Pradyumansinh Jadeja
---
```

Bag of Words model

- Before we can perform analysis on textual data, we must tokenize every word within the dataset.
- The bag of Words model is used to preprocess the text by converting it into a bag of words, which keeps a count of the total occurrences of most frequently used words.
- This model can be visualized using a table, which contains the count of words corresponding to the word itself.
- A bag of words is a representation of text that describes the occurrence of words within a document.
- We just keep track of word counts and disregard the grammatical details and the word order.
- It is called a "bag" of words because any information about the order or structure of words in the document is discarded.
- The model is only concerned with whether known words occur in the document, not wherein the document.
- With the bag-of-Words technique, we can convert variable-length texts into a fixed-length vector.