01 Python math modules

#Python/Advanced

- Python has a built-in math module.
- We can easily calculate many mathematical calculations in Python using the Math module.
- Note - Result of functions is float data type

- Python Math Module

Function	Description
sqrt(n)	Will return square root of given number
pow(n1, n2)	will return n1**n2
ceil(x)	will return lowest integer bigger than or equal to x is returned.
floor(x)	will return lowest integer lesser than or equal to x is returned.
factorial(x)	provides factorial of x
fabs(x)	gives x's absolute value back.
pi	This is constant. return pi value = 3.14
e	e is a constant in mathematics (2.71828...)

- e.g.

```
# This file provides demo of math module
import math
from math import *
```

```
n1, n2 = 5, 144
n3, n4, n5 = 5.02, 5.72, 5.50
n6, n7, n8 = -0.13, 39, -23

print(f"Square of {n1}: {math.pow(n1, 2)}")
print(f"Square root of {n2}: {math.sqrt(n2)}")
print(f"Factorial of {n1}: {math.factorial(n1)}")

print(f"ceil of {n3}: {math.ceil(n3)}")
print(f"ceil of {n4}: {math.ceil(n4)}")
print(f"ceil of {n5}: {math.ceil(n5)}")
print(f"ceil of {n6}: {math.ceil(n6)}")


print(f"floor of {n3}: {math.floor(n3)}")
print(f"floor of {n4}: {math.floor(n4)}")
print(f"floor of {n5}: {math.floor(n5)}")
print(f"floor of {n6}: {math.floor(n6)}")


print(f"absolute value of {n6}: {math.fabs(n6)}")
print(f"absolute value of {n7}: {math.fabs(n7)}")
print(f"absolute value of {n8}: {math.fabs(n8)}")

print("value of pi: ", math.pi)
```

```
Square of 5: 25.0
Square root of 144: 12.0
Factorial of 5: 120
ceil of 5.02: 6
ceil of 5.72: 6
ceil of 5.5: 6
ceil of -0.13: 0
floor of 5.02: 5
floor of 5.72: 5
```

```
floor of 5.5: 5
floor of -0.13: -1
absolute value of -0.13: 0.13
absolute value of 39: 39.0
absolute value of -23: 23.0
value of pi:  3.141592653589793
```

02 Python OS module

#Python/Advanced

- provides the facility to establish the interaction between the user and the operating system.
- It offers many useful OS functions that are used to perform OS-based tasks and get related information about operating system.

How to use OS module

```
import os
```

Function	Use
os.name()	To get name of underlying operating system
os.mkdir(str)	To create directory at OS
os.makedirs(str)	to create directory recursively
os.getcwd()	To get current working directory
os.chdir(path_of_dir)	to change the current working directory
os.rmdir(path_of_dir)	removes the specified directory with an absolute or related path.

os.rename(existing_dir, new_dir_name)	To rename dir or file
os.popen(file_name, mode)/ open()	opens a file or from the command specified, and it returns a file object which is connected to a pipe.
os.close(file_path) / file_obj.close()	to close object already connected with pipe
os.access(file_name, access_to_check)	test if the invoking user has access : a. file path : os.F_OK b. read file : os.R_OK c. write into file: os.W_OK d. execute file: os.X_OK

- e.g.

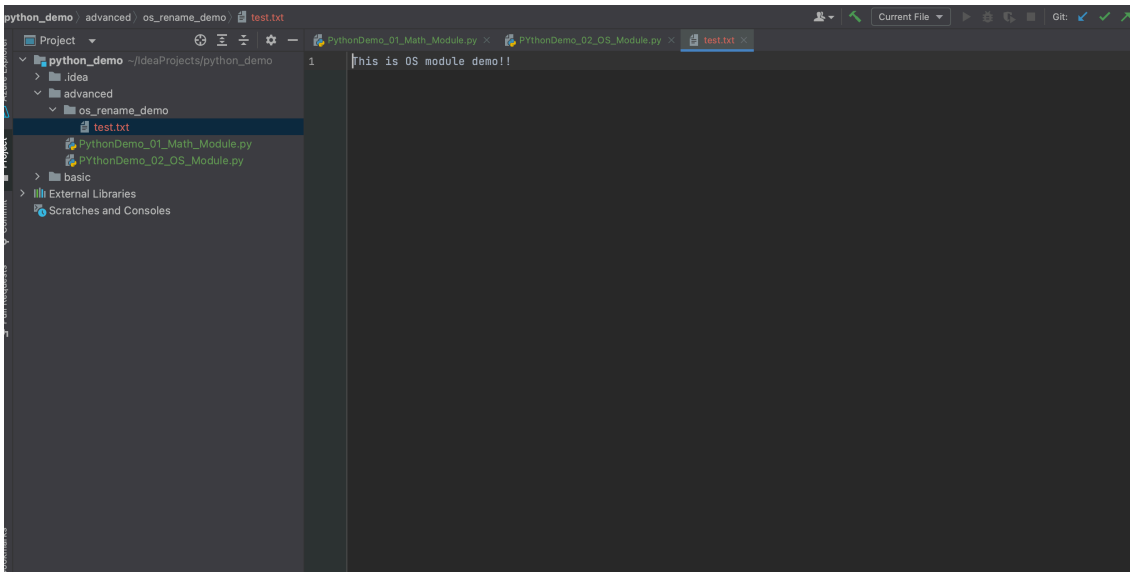
```
# This file provides demo of OS module

import os

file_path = ""


def getDetails():
    # to get OS name
    print(f"OS name: {os.name}")

    # to get CWD
    print(f"Current working dir: {os.getcwd()}")


def createDir():
    global file_path
    file_path = os.getcwd() + "/os_demo/"
    is_dir_exist = os.access(file_path, os.F_OK)
```

```
    print("directory {} exist:
{}".format(file_path, is_dir_exist))
    # to create new dir
    if not is_dir_exist:
        print(f"Creating dir {file_path}")
        os.mkdir(file_path)


def deleteDir():
    global file_path
    print(f"Deleting dir {file_path}")
    os.rmdir(file_path)


def read_or_write(operation_name):
    global file_path

    if operation_name == "w":
        file = open(file_path + "test.txt", "w")
        print("File obj:", file)
        print("Writing info into file..")
        file.write("This is OS module demo!!")
        file.close()
    else:
        file = open(file_path + "test.txt", "r")
        print(file.__dict__)
        print("File obj:", file)
        print("Reading info from file..")
        info = file.read()
        print(info)
        file.close()


def checkAccess():
    global file_path
    txt_file_path = file_path + "test.txt"
    print(f"Access to file path {txt_file_path}:
{os.access(txt_file_path, os.F_OK)}")
    print(f"Access to read file {txt_file_path}:
```

```
{os.access(txt_file_path, os.R_OK)}")
    print(f"Access to write into file
{txt_file_path}: {os.access(txt_file_path,
os.W_OK)}")
    print(f"Access to execute file {txt_file_path}:
{os.access(txt_file_path, os.X_OK)}")


def renameDir():
    global file_path
    # renaming existing dir
    print(f"Existing dir: {file_path}")
    new_dir = file_path.replace("os_demo",
"os_rename_demo")
    print(f"New dir: {new_dir}")
    os.rename(file_path, new_dir)


def main():
    getDetails()
    createDir()
    read_or_write("r")
    checkAccess()


if __name__ == "__main__":
    main()
```

```
OS name: posix
Current working dir: /Users/rohitphadtare/
IdeaProjects/python_demo/advanced
directory /Users/rohitphadtare/IdeaProjects/
python_demo/advanced/os_demo/ exist: True

{'mode': 'r'}
File obj: <_io.TextIOWrapper name='/Users/
```

```
rohitphadtare/IdeaProjects/python_demo/advanced/
os_demo/test.txt' mode='r' encoding='utf-8'>

Reading info from file..
This is OS module demo!!

Access to file path /Users/rohitphadtare/
IdeaProjects/python_demo/advanced/os_demo/test.txt:
True
Access to read file /Users/rohitphadtare/
IdeaProjects/python_demo/advanced/os_demo/test.txt:
True
Access to write into file /Users/rohitphadtare/
IdeaProjects/python_demo/advanced/os_demo/test.txt:
True
Access to execute file /Users/rohitphadtare/
IdeaProjects/python_demo/advanced/os_demo/test.txt:
False

Existing dir: /Users/rohitphadtare/IdeaProjects/
python_demo/advanced/os_demo/
New dir: /Users/rohitphadtare/IdeaProjects/
python_demo/advanced/os_rename_demo/
```



03 Numpy

#Python/Advanced/Numpy

- **Numpy is python library used for working with arrays.**
- **Why to use Numpy?**
    - It is faster as compared to python traditional lists or arrays from array module
        - How it is faster than lists?
            - It stores array objects at one place with continuous memory locations where list does not stores elements at continuous memory locations.
            - This behaviour is called locality of reference in computer science.

- **Installation of Numpy -**
    - `pip install numpy`
    - PIP is a package manager for Python packages, or modules if you like.
        - Package - A package contains all the files you need for a module.
        - how to check pip installed or not -
            - `pip --version`
        - **Note - 'pip', refers to Python 2. pip3 refers to Python 3.**

- **Get Start with Numpy -**
    - How to use Numpy -
        - `import numpy as np`
        - e.g.

```
import numpy as np
arr = np.array(['Zurich', 'Basel', 'Bern',
       'Lausanne', 'Lugano', 'Lucerne'])
print("Type of obj:", type(arr), " Details :
       ", arr)
print("Dimension of array: ", arr.ndim)
```

```
print("Accessing elements of array using
       index")
for i in range(0, len(arr)):
    print(f"arr[{i}]: {arr[i]}")
```

```
Version of numpy 1.26.4
Type of obj: <class 'numpy.ndarray'>
       Details :  ['Zurich' 'Basel' 'Bern'
       'Lausanne' 'Lugano' 'Lucerne']
Dimension of array:  1
Accessing elements of array using index
arr[0]: Zurich
arr[1]: Basel
arr[2]: Bern
arr[3]: Lausanne
arr[4]: Lugano
arr[5]: Lucerne
```

- Array object in Numpy is called as '**ndarray**'. Hence, **array() function of numpy return ndarray object.**
  - We can pass list, tuple or array like object to create ndarray
- **Types of array**
  - **One dimensional array**
    - Array with elements of same data type i.e. uni-dimensional
    - e.g. `arr = np.array((1,2,3))`
  - **Two dimensional array**
    - Array where elements are stored in grid structure like rows and columns i.e. elements stored in two dimensional way
    - Elements of 2-D Array is 1-D array
  - **Three dimensional array**
    - An array that has 2-D arrays (matrices) as its elements is called 3-D array.

- **N-Dimension array**
  - Array that had (n-1)-D arrays as its elements

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically).
Changing the size of an ndarray will create a new array and delete the original.
The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory.

Creating and Initialising arrays

- **Creating empty array -**
  - We can use functions numpy built in functions to create empty arrays with or without value
  - **Note on order -**
    - 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest.
    - 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest.
    - 'K' - keep order (default when creating new array based on existing one)
    - 'A'- any order (Try not to use)
  - **dtype** - data type of elements in array
    - Valid types -
      - Numeric -
        i - integer
        b - boolean
        u - unsigned integer
        f - float
        c - complex float
      - Date and time -
        m - timedelta
        M - datetime

- Strings -
    - O - object (**useful for variable string items in array**)
    - S - string
    - U - unicode string
- V - fixed chunk of memory for other type (void)

- **shape** - it gives idea about in how many dimensions data is stored in array.
    - e.g.
        - if shape is 5 then array is one dimensional with length 5
        - if shape [2, 5] then array is two dimensional with two rows and 5 columns
        - if shape [2, 2, 5] then array is three dimensional with three rows where every rows contains 2-D array of 2 rows and 5 columns

Function	**Use of function**
numpy.empty(shape_of_array, data_type_of_elements_in_array , order - default 'C' (either use 'C' or 'F'))	To create empty array of given shape and objects. **Note** - It does not set the array values to zero, and may therefore be marginally faster.
numpy.zeros(shape,dtype)	Create array with initial values of every memeber as 0
numpy.ones(shape,dtype)	Create array with initial values of every memeber as 1
numpy.full(shape, fill_value, dtype)	Create array with initial values of every memeber as fill_value
numpy.empty_like(existing_arr, dytpe, order, shape)	Return a new array with the same shape and type as a given array.

numpy.zeros_like(existing_arr, dytpe, order, shape)	Return a new array with zero values with the same shape and type as a given array.
numpy.ones_like(existing_arr, dytpe, order, shape)	Return a new array with ones of the same shape and type as a given array.
numpy.full_like(existing_arr, fill_value, dytpe, order, shape)	Return a new array with fill_values of the same shape and type as a given array.

e.g.

```
arr = np.zeros(5, 'i')
print("Empty arr: ", arr, " dtype:", arr.dtype, "
dimension: ", arr.ndim)

str_arr = np.zeros(5, "O")
print("Empty string arr: ", str_arr, " dtype:",
str_arr.dtype, " dimension: ", str_arr.ndim)

arr = np.ones(5, 'i')
print("Array with ones : ", arr, " dtype:",
arr.dtype, " dimension: ", arr.ndim)

str_arr = np.full(5, "Test", object)
print("String arr with default values: ", str_arr,
" dtype:", str_arr.dtype, " dimension: ",
str_arr.ndim)

zero_like_arr = np.zeros_like(arr)
print("zero_like_arr: ", zero_like_arr, " dtype:",
      zero_like_arr.dtype, " dimension: ",
zero_like_arr.ndim)

one_like_arr = np.ones_like(str_arr, dtype='f',
shape=2)
print("one_like_arr: ", one_like_arr, " dtype:",
```

```
one_like_arr.dtype,
      " dimension: ", one_like_arr.ndim)

full_like_arr = np.full_like(one_like_arr,
fill_value="Hi", dtype=object, shape=[1, 2])
print("full_like_arr: ", full_like_arr, " dtype:",
full_like_arr.dtype,
      " dimension: ", full_like_arr.ndim)
```

Output

```
Empty arr:  [0 0 0 0 0]  dtype: int32  dimension:
1
Empty string arr:  [0 0 0 0 0]  dtype: object
dimension:  1
Array with ones :  [1 1 1 1 1]  dtype: int32
dimension:  1
String arr with default values:  ['Test' 'Test'
'Test' 'Test' 'Test']  dtype: object  dimension:  1
zero_like_arr:  [0 0 0 0 0]  dtype: int32
dimension:  1
one_like_arr:  [1. 1.]  dtype: float32  dimension:
1
full_like_arr:  [['Hi' 'Hi']]  dtype: object
dimension:  2
```

Indexing and accessing arrays

- ## Indexing
  - ### 1-D array indexing
    - e.g.
      - `arr[0]` → this will give 1st element
      - `arr[1] = 1` → this will assign value at 2nd position in array
      - `arr[1:3] = 2` → this will assign value at 2nd and 3rd position in array

- **2-D array indexing**
  - e.g. `arr[0, 1]` → this will give 1st element from 1st row and 1st column

```
arr = np.array([[1, 2], [3, 4], [5, 6], [7,
       8], [9, -1]], dtype='i')

print("element at 1st row and 2nd column: ",
       arr[1, 1])


print("update value at 1st row and 2nd column
       to 9")
arr[1, 1] = 9
print("Array post updating element at 1st row
       and 2nd column")
print(arr)


print("update value at 2nd row: 7")
arr[2:3] = 7
print("Array post updating element at 3rd
       row")
print(arr)
```

```
element at 1st row and 2nd column:  4

update value at 1st row and 2nd column to 9
Array post updating element at 1st row and
       2nd column
[[ 1  2]
[ 3  9]
[ 5  6]
[ 7  8]
[ 9 -1]]
```

```
update value at 2nd row: 7
Array post updating element at 3rd row
[[ 1  2]
[ 3  9]
[ 7  7]
[ 7  8]
[ 9 -1]]
```

- **Slicing**
  - Slicing will provide new array from existing array basis of start and end index position with skip value
    - **1-D array slicing**
      - e.g.
      - To get new array - `arr[2:6]` → this will give new array from 3rd element till 6th element
      - To get new array - `arr[2:6:2]` → this will give new array from 3rd element till 6th element. However, it will return elements from 2nd and 4th position as we have mentioned skip value as 2
        - it will pick up alternate elements from mentioned start and end position in slicing
      - by default start index = 0, end index = last element of array
      - we can use negative slicing as well
        - e.g.
          - Assume arr = [1,2,3,4,5,6]
          - `arr[-3:-1]` → this will give array(4,5)
          - **-1 is last element of array**

```
        def one_dimensional_array():
            num_list = random.sample(range(0, 100),
7)
            arr = np.array(num_list, 'i')
            print("Accessing elements using
indexing of one dimensional array")
```

```
            for i in range(arr.size):
                print(f"arr[{i}]: {arr[i]}",
end="\t")

            print("")
            print("Updating value at 2nd position
to 39")
            arr[1] = 39
            print("Post updating value at 2nd
position: ", arr)

            print("Updating values at more than one
position using slicing "
                  " i.e. position 6 and 7 ")
            arr[5:7] = 29
            print("Post updating value at 6th and
7th position: ", arr)

            print("Array from 2nd element till 7th
element "
                  "with skip = 2 "
                  "using slicing: ", arr[1:6:2])

            print("To access array using negative
indexing ", arr[-3:])

            print("To reverse array ", arr[::-1])
```

```
        Accessing elements using indexing of one
dimensional
        array
        arr[0]: 45  arr[1]: 93   arr[2]: 65
   arr[3]: 85   arr[4]: 86   arr[5]: 24
   arr[6]: 49

        Updating value at 2nd position to 39
```

```
        Post updating value at 2nd position:  [45
39 65 85 86 24 49]

        Updating values at more than one position
using slicing  i.e. position 6 and 7
        Post updating value at 6th and 7th
position:
        [45 39 65 85 86 29 29]

        Array from 2nd element till 7th element
with skip = 2 using slicing:  [39 85 29]
        To access array using negative indexing
[86 29 29]
        To reverse array  [29 29 86 85 65 39 45]
```

- **2-D array slicing**
  - e.g.

```
        arr = np.zeros((2, 2), dtype='i')
        print("2-D array with zero values ", arr)


        arr = np.array([[1, 2], [3, 4], [5, 6], [7,
                  8], [9, -1]], dtype='i')
        print("2-D array with initial values ",
                  arr, " dimension: ", arr.ndim,
              " Data type: ", arr.dtype, " size of
                  array: ", arr.size,
              "shape: ", arr.shape)

        new_arr = arr[1:4:2]
        # get 2-D array from existing one
        print("Getting new 2-D array from existing
                  one using"
              " slicing", new_arr, " shape: ",
                  new_arr.shape)

        new_arr = arr[1:4:2, 0:1]
```

```
# get 2-D array from existing one with one
        column
print("Getting new 2-D array with one
        column "
      "from existing one using"
      " slicing", new_arr, " shape: ",
        new_arr.shape)
```

```
2-D array with zero values  [[0 0]
 [0 0]]

2-D array with initial values  [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 -1]]  dimension:  2  Data type:  int32
size of array:  10 shape:  (5, 2)

Getting new 2-D array from existing one
        using slicing [[3 4]
 [7 8]]  shape:  (2, 2)

Getting new 2-D array with one column from
        existing one usingslicing
[[3]
 [7]]
shape:  (2, 1)
```

- **Copy and View method**
  - **Copy**
    - We can create new array using this method from existing one
    - Here, we get copy of original array. Hence, any modifications to this copy doesn't reflect any changes to original one

- **View**
  - We can create new array using this method from existing one. **Only key difference is this return a view i.e. any modifications to this will result in changes to original array from which it is derived**
- **Base**
  - This parameter of ndarray object gives whether array is view or not
    - If value is none, it means ndarray is not a view
    - If value is not none, it means ndarray is a view
- e.g.

```
def copy_and_view_demo():
   arr = np.array([1, 2, 3, 4], dtype=int)
   print("Original arr : ", arr, " base: ",
    arr.base)
   arr1 = arr.copy()
   print("Copied arr : ", arr1, " base: ",
    arr1.base)
   arr2 = arr.view()
   print("Viewed arr : ", arr2, " base: ",
    arr2.base)

   print("updating element at 1st index "
 "in copied array with value 5 and"
 "view with value 7")

 arr1[1] = 5
   arr2[2] = 7

 print("post update")
 print(f"original array: {arr}")
 print(f"Copied array: {arr1}")
 print(f"view: {arr2}")
```

```
Original arr :  [1 2 3 4]  base:  None
Copied arr :  [1 2 3 4]  base:  None
Viewed arr :  [1 2 3 4]  base:  [1 2 3 4]
updating element at 1st index in copied array
    with value 5 andview with value 7
post update
original array: [1 2 7 4]
Copied array: [1 5 3 4]
view: [1 2 7 4]
```

- ## Shaping and Reshaping
  - **The shape of an array is the number of elements in each dimension.**
  - NumPy arrays have an attribute called **shape** that returns a tuple with each index having the number of corresponding elements.
  - ## Shaping
    - **How to get shape of array -**
      - `arr.shape()`
        This will return tuple
        e.g.
        - result of above statement is (3,). It means array is one dimensional with size 3
        - result of above statement is (3, 2). It means array is two dimensional. i.e. array has 2 dimensions, where the first dimension has 3 elements and the second has 2.
    - How to create array with shape -

```
def shapeDemo():
    arr = np.full((3, 2), 1, dtype='i')
    for i in arr:
        for j in i:
            print(j, end="\t")
```

```
            print("")
```

```
1   1
1   1
1   1
```

- **Reshaping**
  - Reshaping means changing the shape of an array.
  - By reshaping we can add or remove dimensions or change number of elements in each dimension.
  - **How to reshape ?**
    - **by using 'reshape()' method**

      note - It returns view

      e.g.

```
        def reShapeDemo():
            num_list = random.sample(range(1, 100),
6)
            arr = np.array(num_list, dtype='i')

            print(f"Input array: {arr} with shape:
{arr.shape} and base: {arr.base}")
            reshaped_arr_1 = arr.reshape(2, 3)
            print(f"Reshaped array:
{reshaped_arr_1} "
                  f"with shape:
{reshaped_arr_1.shape} "
                  f"and base:
{reshaped_arr_1.base}")
```

```
Input array: [30 95 38 21 15 98] with
```

```
shape: (6,) and base: None

        Reshaped array: [[30 95 38] [21 15 98]]
        with shape: (2, 3) and base: [30 95 38 21
15 98]
```

what if size of existing array is odd or not able to match with new shape

- we can use '**resize**' to resize that array with such dimension which can be useful for reshaping it
- e.g.

```
          def reShapeDemo():

              num_list = random.sample(range(1,
100), 7)
              arr = np.array(num_list, dtype='i')
              print(f"Input array before resize:
{arr}
                    with shape: {arr.shape} and
base: {arr.base}")

              try:
                  reshaped_arr = np.reshape(arr,
newshape=(2, 4))

              except ValueError as v:
                  print("Error received : ", v)
                  print("Need to resize now input
array !!!")

                  arr = np.resize(arr, 8)

                  print(f"Input array post resize:
{arr}
```

```
                    with shape: {arr.shape} and
base: {arr.base}")

                reshaped_arr = np.reshape(arr,
newshape=(2, 4))
                print(f"Reshaped array with new
shape: {reshaped_arr.shape}
                           and base:
{reshaped_arr.base}")
                for i in reshaped_arr:
                    for j in i:
                        print(j, end="\t")
                    print("")

            finally:
                print("-"*50)
```

```
          Input array before resize: [51  1 42 11
25 59 19]
          with shape: (7,) and base: None

          Error received :  cannot reshape array of
size 7 into shape (2,4)
          Need to resize now input array !!!
          Input array post resize: [51  1 42 11 25
59 19  0]
          with shape: (8,) and base: None

          Reshaped array with new shape: (2, 4)
          and base: [51  1 42 11 25 59 19  0]
          51  1   42  11
          25  59  19  0

--------------------------------------------------
```

By using flatten and ravel -

- If we want to convert multi-dimensional array to one-dimensional we can use flatten() or ravel()
- **key difference - flatten() will give new ndarray object. On the other hand, ravel() will give view**
- e.g.

```
        def reShapeDemo2():
            arr = np.array([[1, 2, 3], [4, 5,
6]], dtype='i')
            print(f"Input array before reshape:
{arr} with shape: {arr.shape} and base:
{arr.base}")
            arr1 = arr.flatten()
            arr2 = arr.ravel()
            print("Flatten existing array: ",
arr1, " with base: ", arr1.base)
            print("ravel existing array: ", arr2,
" with base: ", arr2.base)
```

```
        Input array before reshape: [[1 2 3] [4 5
6]]
        with shape: (2, 3) and base: None

        Flatten existing array:  [1 2 3 4 5 6]
with base:  None

        ravel existing array:  [1 2 3 4 5 6]
         with base:  [[1 2 3] [4 5 6]]
```

- **Iterating through ndarray -**
  - **using simple loops -**

```
arr = np.array([[1, 2, 3], [4, 5, 6]],
    dtype='i')

# iterating using loops
for x in arr:
    for y in x:
        print(y, end="\t")
    print("")
```

  - **using nditer() method -**
    - it is advanced iterator
    - using this we can iterate on single element of multiple dimensional array
    - we can iterate over elements using different datatypes
    - we can use different step size as well
    - e.g

```
arr = np.array([[1, 2, 3], [4, 5, 6]],
      dtype='i')
for x in np.nditer(arr[::, :2],
      flags=['buffered'], op_dtypes='S'):
    print(x)
```

```
b'1'
b'2'
b'4'
```

```
  b'5'
```

- **Joining array -**
  - Joining means putting contents of two or more arrays in a single array.
  - Methods to join -
  - concatenate((arr_1, arr_2), axis)
  - stack(arr_1, arr_2, axis) - to stack along axis
  - hstack(arr_1, arr_2) - to stack along rows
  - vstack(arr_1, arr_2) - to stack along columns

```
def joinDemo():
    arr1 = np.array([1, 2, 3, 4, 5], dtype='i')
    arr2 = np.array([7, 8, 9, 10, 11],
     dtype='i')

    # using concatenate
    print(np.concatenate((arr1, arr2)))

    # using stack
    print(np.stack((arr1, arr2), axis=1))

    # using hstack
    print(np.hstack((arr1, arr2)))

    # using vstack
    print(np.vstack((arr1, arr2)))
```

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

[
 [ 1  7]
```

```
 [ 2  8]
 [ 3  9]
 [ 4 10]
 [ 5 11]
]

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

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

- **Spliting array -**
  - use array_split()
  - NumPy Splitting Array

- **Sorting -**
  - use np.sort() method
  - **Note:** This method returns a copy of the array, leaving the original array unchanged.

- **Searching -**
  - we can use **where**() method
    - e.g `x = np.where(arr == 4)`
    - this will return tuple with indexes where value '4' is present in array

  - **searchsorted() method -**
    - **returns the index where the specified value would be inserted to maintain the search order.**
    - e.g.

```
import numpy as np
```

```
arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7)

print(x)



Output - 4
```

- **Filtering**
  - In NumPy, you filter an array using a *boolean index list*.
  - NumPy Filter Array

Important Links
numpy.append — NumPy v1.26 Manual
NumPy ufuncs - Logs

3.1 Numpy other functions

#Python/Advanced/Numpy

- **Set Operations**
  - **Unique**() - to get array with unique elements
  - **union1d**(arr1, arr2) - find unique values of two arrays and return 1d array
  - **intersect1d**(arr1, arr2) - To find only the values that are present in both arrays
  - **setdiff1d**(arr1, arr2) - To find only the values in the first set that is NOT present in the seconds set

    - **setxor1d**(arr1, arr2) - To find only the values that are NOT present in BOTH sets

- LCM -
    - To find the Lowest Common Multiple of all values in an array
    - np.lcm.reduce(arr)
    - e.g. `np.lcm.reduce([3, 6, 9])` → 18

- GCD -
    - To find the Highest Common Factor of all values in an array
    - np.gcd.reduce(arr)
    - e.g. `np.gcd.reduce([3, 6, 9])` → 3

3.2 Numpy Random

#Python/Advanced/Numpy

- This module help to generate array with integers, floats

- To generate random int -
    - `random.randint(low, high, shape)`

- To generate random float -
    - `random.rand(d1..dn)` → generate random float between 0 and 1
    - To generate random float between range
        - `random.uniform(low,high,shape)`

- Generate array of random integers -
    - one-dimensional
    - two-dimensional
    - use `random.randint(low, high, shape)`

- Generate array using existing elements from list -

    - choice method
    - choice with probability option
    - e.g. `random.choice([10, 20, 30], 6, p=[0.25, 0.45, 0.30]))`
        - `random.choice([10.5, 12.0, 11.25, 31.5], 3))`

- Shuffle elements of existing array -
    - random.shuffle(arr)
    - This will change original array

- Permutation of existing array -
    - This will provide new array from existing one which is one probability sample of shuffling
    - This will not change original array
    - random.permutation(arr)

04 Pandas - Introduction to Series

#Python/Advanced/Pandas

Important Links
Pandas Tutorial
Getting started — pandas 2.2.1 documentation

- It is python library used to analyse data
- Pandas stands for Panel Data and Python Data Analysis

- **How to install pandas**
    - `pip3 install pandas`

    - How to upgrader pip version - `pip3 install --upgrade pip`

- **How to use pandas**
    - `import pandas as pd`

- # Pandas Series
  - It is nothing but column in table.
    - It is one dimensional array which holds data of any type
    - e.g.

```
import pandas as pd


a = [1, 2.2, "Pooja"]
s1 = pd.Series(a)
print(s1)
```

```
0         1
1       2.2
2    Pooja
dtype: object
```

  - ## Labels -
    - Labels are used to access specific values from series.
    - As series are nothing but array, we can access values in it using index positions. **Those index positions are nothing but Labels**
    - e.g. `print(s1[2])` → `Pooja`

    - We can assign name to labels using **index** argument and hence, we can access elements in series using Labels with name.
    - e.g.

```
a = [1, 2.2, "Pooja"]
s1 = pd.Series(a, index=["id", "module_num",
       "name"])
print(s1['module_num'])
```

Output - `2.2`

- **If we create series using dictionary then dictionary key's become Series Label and dictionary value becomes values in Series**
- e.g.

```
d1 = {'c1': 'India', 'c2': 'USA', 'c3':
    'Switzerland'}
print("Dictionary: ", d1)
s2 = pd.Series(d1)
print("Series: ")
print(s2)
```

Output -

```
Dictionary:  {'c1': 'India', 'c2': 'USA', 'c3':
    'Switzerland'}
Series:
c1          India
c2            USA
c3    Switzerland
dtype: object
```

- ## Series Operation's
  - **Access value from series -**
    - Using index position, names and slice operator
    - e.g. `print(s1[0:1])` → id 1
  - **Filter values from series on basis of condition -**
    - Use `loc[series_var_name expression_to_find]` or

```
series_var[series_var
expression_to_find]
```

- e.g.

```
print("Index where value 4 is
present : \n ", s5[s5 == 4])

print("Index where value 4 is
present : \n ", s5.loc[s5 ==
4])
```

- **Update value from series -**
  - We can use `iloc` or `loc` function to update existing values in Series

    Difference between iloc and loc - iloc only used index position.However, loc uses index as well as labels for accessing elements. Also, slicing is only allowed in iloc
  - e.g.

```
a = [1, 2.2, "Pooja", 4, 5]
s1 = pd.Series(a, index=["id",
          "module_num", "name", "n1", "n2"])
print("Series before update\n", s1)
s1.loc[0] = 2
s1.iloc[3:5] = 12
s1['name'] = 'Mrs. Pooja'
print("Series before update")
print(s1)
```

Output

```
Series before update
 id                1
module_num      2.2
```

```
name           Pooja
n1                 4
n2                 5
dtype: object

  Series after update
  id                     2
  module_num           2.2
  name          Mrs. Pooja
  n1                    12
  n2                    12
```

- We can use other methods like `s1._set_value(label_name, value)`

- **Add new element is series -**
  - We don't have any function which adds one element to existing series. However, we have `append` function which generally appends list or tuple to existing series.
  - e.g.

```
arr = [1, 2, 3]
arr2 = [10, 20, 30]
s3 = pd.Series(arr)
s4 = pd.Series(arr2)
s5 = s3._append(s4, ignore_index=True)
print("S4 series \n", s5)
```

Output -

```
S4 series
 0     1
1     2
2     3
```

```
3    10
4    20
5    30
```

- **Note - _append is not recommended as its original method append was discontinued in 2.0. So always use pd.concat() instead to append multiple Series together**
- e.g. `s5 = pd.concat([s3, s4], ignore_index=True)`

- **Delete elements from Series -**
  - We can use `drop()` functions to delete elements on basis of index or labels
  - **Note - By default this method does not change original Series object. Instead it will provide new Series object. To avoid this we can use argument series_name.drop(inplace=True)**
  - **Delete single element using labels -**

    ```
    print(s5.drop(labels=0))
    ```

    Output -

```
1     2
2     3
3    10
4    20
5    30
dtype: int64
```

  - **Delete multiple elements using label index -**

    ```
    print(s5.drop(index=[0, 2]))
    ```

    Output -

```
1     2
3    10
4    20
5    30
dtype: int64
```

- **Delete multiple elements using labels -**
  `print(s2.drop(labels=['c1', 'c2']))`
  Output -

```
c2    USA
dtype: object
```

- **Give name to series -**
  - Using **name** attribute
  - `s2 = pd.Series(d1, name='Country')`
  - `s5.name = "s5"`
  - Similarly series is having other attributes like **dtype** - to get data type of elements present in series, **size** - to get size of series

- **Other operations**
  - Important Link -
    pandas.Series — pandas 2.2.1 documentation

4.1 Pandas - DataFrame

#Python/Advanced/Pandas

- A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.
- **Create DataFrame**
  - Using `DataFrame()` function
  - e.g.

```
d1 = {
    "id": [1, 2, 3],
    "name": ["Rohit", "Pooja", "Rajani"]
}
print(f"Type of d1: {type(d1)}")
df = pd.DataFrame(d1)
print(df)
```

Output -

```
Type of d1: <class 'dict'>
   id    name
0   1   Rohit
1   2   Pooja
2   3  Rajani
```

- Important attributes from `DataFrame()` functions -
  - index - to provide label values
  - column - to provide column name
  - e.g.

```
d2 = [10, 20, 30]
print(pd.DataFrame(d2, index=['first',
          'second', 'third'],
          columns=['id']))
```

```
        id
first   10
second  20
third   30
```

- Using **concat** method -

- e.g.

```
d3 = [random.randint(i, i*10) for i in
       range(0, 500)]
d4 = ['A', 'B', 'C', 'D', 'E']*100

s3, s4 = pd.Series(d3), pd.Series(d4)

df1 = pd.concat([s3, s4],
       axis=1).rename(columns={0: 'id', 1:
       'str_val'})
```

- **Access rows from data frame**
  - **To access details of data frame - info()**
    - e.g. `print(df.info())` gives below output

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      3 non-null      int64
 1   name    3 non-null      object
dtypes: int64(1), object(1)
memory usage: 180.0+ bytes
None
```

  - To print data frame - `print(df)`
    - By default it will display all rows if data frame size is less than pd.options.display.max_rows
    - if it is greater than pd.options.display.max_rows value then it will display first and last 5 rows with headers
  - Access elements from starting position - `df.head(n)`

- Access elements from starting position - `df.tail(n)`
  - By default 5 rows will display with headers for head and tail
  - We can specify number of rows we want from head or tail function

- **Access rows using loc or iloc**
  - We can use index values or label name in `loc` However, in `iloc` we only use index positions.
  - For single value
    - `loc[0]`
      Remember this will only work if index with labels are not present. If labels are present then we must use `iloc[0]`
    - using label name - `loc['label_name']`
  - For multiple values
    - `loc[[0,1]]` or `iloc[[0, 2]]`
    - with labels - `print(df2.loc[['first', 'second']])`
    - slicing -
      using indexes - `df2.iloc[:2]`
      - Note - when we are using slicing with `iloc` it will consider last element as last position index -1. However, `loc` will consider last element as last position index mentioned in slicing.

      using labels -
      `df2.loc['third':'fourth']`
      - Note in slicing with labels it will include end position as well. Hence, in above example 'fourth' value will be there
  - e.g.

```
d2 = [10, 20, 30, 40]
df2 = pd.DataFrame(d2, index=['first',
       'second', 'third', 'fourth'],
```

```
        columns=['id'])
print("using iloc with index \n",
        df2.iloc[1])
print("using loc with label index \n",
        df2.loc['fourth'])
print("/"*15)

print("using iloc with multiple rows \n",
        df2.iloc[[0, 2]])
print("using loc with multiple rows \n",
        df2.loc[['second', 'fourth']])
print("/"*15)

print("using iloc slicing \n", df2.iloc[:2])
print("using loc slicing \n",
        df2.loc['third':'fourth'])
print("-"*25)
```

Output -

```
using iloc with index
 id    20
Name: second, dtype: int64

using loc with label index
 id    40
Name: fourth, dtype: int64
///////////////

using iloc with multiple rows
        id
first  10
third  30

using loc with multiple rows
         id
second  20
fourth  40
```

```
///////////////

using iloc slicing
         id
first   10
second  20

using loc slicing
         id
third   30
fourth  40
-------------------------
```

- **Select multiple columns from data frame -**
  - using column names in subscript operator or we can use `loc` as well
  - e.g.

```
df1 = pd.concat([s3, s4],
    axis=1).rename(columns={0: 'id', 1:
    'str_val'})
print(df1[['id', 'str_val']][1:5])
print("-"*25)
```

Output -

```
   id str_val
1   6       B
2   6       C
3  14       D
4  10       E
-------------------------
```

  - e.g. for `loc`

        - `print(df1.loc[1:3, ['id', 'name']])`

- **Select data from data frame on basis of criteria -**
    - **Using index and column name -**
        - Traverse using index and filter using column value
        - e.g.

```
d3 = [random.randint(i, i*10) for i in
       range(0, 500)]
d4 = ['A', 'B', 'C', 'D', 'E']*100

s3, s4 = pd.Series(d3), pd.Series(d4)

df1 = pd.concat([s3, s4],
       axis=1).rename(columns={0: 'id', 1:
       'str_val'})

for x in df1.index:
    if df1.loc[x, 'id'] % 500 == 0:
        print(df1.loc[x]['id'])
```

    - **Filter using single column value in loc function -**

```
print("Single column condition: \n",
    df1.loc[df1['str_val'] == 'A'])
```

    - **Filter using multiple column values -**

```
print("Double column condition: \n",
df1.loc[(df1['str_val'] == 'A') & (df1['id'] %
    10 == 0)])
```

        - For ‘OR’ condition we can use - ‘|’

- **Using query function -**

```
print(df1.query("str_val == 'B' and (id > 110
    and id < 700)"))
```

- **Filter function -**
  - This is used to select column names from dataframes
  - `filter(items, like)`
    - items - provide list of columns which need to select
    - like - we can provide string here. Filter will select then only those columns which contains this string in their name
  - e.g.

```
print(df1.filter(items=['id',
       'str_val']).head(20))
```

```
print(df1.filter(like='val').head(20))
```

  - **Note - like, items are mutually exclusive. Means we can use either of them at a time.**

- **where function -**
  - it is used to check a data frame for one or more conditions and return the result accordingly.
  - By default, The rows not satisfying the condition are filled with NaN value.

e.g.

```
d = {
    'id': [1, 2, 3, 4, 5],
    'name': ['R', 'O', 'H', 'I', 'T']
```

```
}
df = pd.DataFrame(d)

cond1 = df['id'] > 2
cond2 = df['name'] != 'T'

print(df.where(cond1 & cond2))
```

Output

```
    id name
0  NaN  NaN
1  NaN  NaN
2  3.0    H
3  4.0    I
4  NaN  NaN
```

- **To find duplicates in data frame -**
  - To get duplicate rows in data frame we can use `duplicated()` function.
    - `duplicated` - will return series of boolean denoting duplicated rows
  - To remove duplicate rows from data frame we can us `drop_duplicates()`
  - e.g.

```
d = {
    'id': [1, 2, 3, 4, 5, 1],
    'name': ['R', 'O', 'H', 'I', 'T', 'R']
}
df = pd.DataFrame(d)
print("Input data frame: ", df)

duplicated_series = df.duplicated()

print("Find duplicate rows \n",
```

```
    duplicated_series)

print("Find only duplicated row \n",
    df[duplicated_series == 1])

print("Remove duplicate rows \n",
    df.drop_duplicates())

print("Remove duplicate on basis of column
    values \n",
    df.drop_duplicates(subset=['name']))
```

Output

```
Input data frame:
  id name
0   1    R
1   2    O
2   3    H
3   4    I
4   5    T
5   1    R

Find duplicate rows
0    False
1    False
2    False
3    False
4    False
5     True
dtype: bool

Find only duplicated row
    id name
5   1    R

Remove duplicate rows
```

```
    id name
0   1    R
1   2    O
2   3    H
3   4    I
4   5    T

Remove duplicate on basis of column values
    id name
0   1    R
1   2    O
2   3    H
3   4    I
4   5    T
```

- How to update duplicate values -
  - e.g.

```
print("update duplicated values in df")
for i in df[duplicated_series ==
       1].index.values:
    df.loc[i] = (6, 'P')
```

- **How to handle null values**
  - Using functions like `isna()`, `notna()`, `dropna()`, `fillna()` functions
  - What is ‘isna()’ and ‘notna()’ -
    - 'isna' - is null any, 'notna' - not null any
    - isna - will give result with boolean true where null is present
    - notna - will give result with boolean true where null is not present
    - Used to check which rows are having null values
    - e.g.

```
d = {
```

```
    'id': [1, 2, 3, 4, 5, 1, None, 8],
    'name': ['R', 'O', 'H', 'I', 'T', 'R',
       'C', None]
}
df = pd.DataFrame(d)

print("Data frame with null")
print(df.isna())
```

Output

```
Data frame with null
      id   name
0  False  False
1  False  False
2  False  False
3  False  False
4  False  False
5  False  False
6   True  False
7  False   True
```

- `dropna()` - to drop those rows which contains null value in either of their columns
  - e.g

```
print("Remove null values from data frame")
df1 = df.dropna()
df2 = pd.concat([pd.to_numeric(df1['id'],
       downcast="integer"), df1['name']],
       axis=1)
print(df2)
```

output

```
Remove null values from data frame
   id name
0   1    R
1   2    O
2   3    H
3   4    I
4   5    T
5   1    R
```

- We can use subset argument in dropna() to drop rows based on null values in selected columns

- `fillna` - To fill null values in rows

e.g.

```
print("Fill null values from data frame")
d1 = {'id': 0, 'name': '#'}
df1 = df.fillna(d1).astype(dtype={'id': int,
    'name': object})
print(df1)
```

Output

```
Fill null values from data frame
   id name
0   1    R
1   2    O
2   3    H
3   4    I
4   5    T
5   1    R
6   0    C
7   8    #
```

- **How to insert rows into data frame**
  - **To add single row -**
    - `df.loc[df.index.max() + 1] = (9, 'R')`
  - **To add multiple rows -**

e.g.

```
multiple_rows = {
    'id': [7, 8, 9, 10, 11, 12],
    'name': ['R', 'A', 'K', 'A', 'S', 'H']
}
df1 = pd.DataFrame(multiple_rows)
df = pd.concat([df, df1], axis=0,
ignore_index=True)
print(df)
```

Output

```
    id name
0    1    R
1    2    O
2    3    H
3    4    I
4    5    T
5    6    P
6    7    R
7    8    A
8    9    K
9   10    A
10  11    S
11  12    H
```

- **How to remove rows from data frame**
  - Use `drop` function
    - Logic : to remove rows we need to know rows at

which index needs to remove

- **Remove rows using indexes -**

e.g.

```
d = {
    'id': [1, 2, 3, 4, 5],
    'name': ['R', 'O', 'H', 'I', 'T']
}
df = pd.DataFrame(d)

index_list = df.index.values

print("Remove first two elements from dataframe
    using indexes")
print(df.drop(index=index_list[:2]))
```

Output

```
Remove first two elements from dataframe using
    indexes
   id name
2   3    H
3   4    I
4   5    T
```

- **Remove rows on basis of condition -**

e.g.

```
print("Remove elements from dataframe on basis
    of condition")
df.drop(df[df['id'] == 2].index.values,
    inplace=True)
```

Output

```
Remove elements from dataframe on basis of
     condition
   id name
0   1    R
2   3    H
3   4    I
4   5    T
```

- **Remove column from data frame**
  - Use `columns` attributes in `drop` function

e.g.

```
print("Remove column id from dataframe")
df.drop(columns={'id'}, inplace=True)
```

Output

```
Remove column id from dataframe
  name
0    R
2    H
3    I
4    T
```

- **How to rename column name in dataframe**
  - Using `rename` function and provide mapper into it (dictionary object with key as old column name and value as new column name)
  - e.g. `dept_renamed_df = dept_df.rename(columns={'dept_id': 'id'})`
- **How to join data frames on basis of column**
  - **Using join function**
    - syntax -

```
df1.join(df2.set_index(col_name_for_jo
in), on='col_name_for_join')
```

- Please remember col_name_for_join must be same in both dataframes. Otherwise, will get key error
- e.g.

```
    emp_data = {
        'emp_id': [10, 20, 30, 40, 50, 60],
        'emp_name': ["Rohit", "Pooja", "Rajani",
"Rushi", "Rutu", "Prithvi"],
        'emp_sal': [5600, 6200, 7900, 7623.45,
5823.41, 5399.14],
        'dept_id': [1, 2, 3, 1, 3, 3]
    }

    dept_data = {
        'dept_id': [1, 2, 3],
        'dept_name': ["IT", "Civil", "Computer
Science"]
    }

    emp_df = pd.DataFrame(emp_data)
    dept_df = pd.DataFrame(dept_data)
    print("Emp df \n", emp_df)
    print("Dept df \n", dept_df)

    print("Joined df")
    print(emp_df.join(dept_df.set_index('dept_id'),
on='dept_id', how='inner'))
```

Output

```
    Joined df
       emp_id emp_name  emp_sal  dept_id
dept_name
    0      10    Rohit  5600.00        1
IT
```

```
    1      20    Pooja  6200.00        2
Civil
    2      30   Rajani  7900.00        3  Computer
Science
    3      40    Rushi  7623.45        1
IT
    4      50     Rutu  5823.41        3  Computer
Science
    5      60  Prithvi  5399.14        3  Computer
Science
```

- Details - pandas.DataFrame.join — pandas 2.2.1 documentation

- Using `Merge` functions -
  - It returned a DataFrame of the two merged objects.
  - Syntax - `df1.merge(df2, how='inner', on=col_name)`

  - When column names are from data frames are not same then we can use ‘left_on’ & ‘right_on’ attributes of `merge` function
  - e.g.

```
print("Merged df when column name is not
      same")
print(emp_df.merge(dept_df, how='left',
      left_on='dept_id', right_on='id'))
```

output

```
Merged df when column name is not same
   emp_id emp_name  emp_sal  dept_id  id
       dept_name
0      10    Rohit  5600.00        1   1
       IT
```

```
1       20    Pooja  6200.00        2   2
        Civil
2       30   Rajani  7900.00        3   3
        Computer Science
3       40    Rushi  7623.45        1   1
        IT
4       50     Rutu  5823.41        3   3
        Computer Science
5       60  Prithvi  5399.14        3   3
        Computer Science
```

- When column name is same then we can use as per easy syntax

e.g.

```
print("Merged df when column names are same")

        print(emp_df.merge(dept_df.rename(colu
        mns={'id': 'dept_id'}), how='left'))
```

output

```
 Merged df when column names are same
   emp_id emp_name  emp_sal  dept_id
        dept_name
0       10    Rohit  5600.00        1
        IT
1       20    Pooja  6200.00        2
        Civil
2       30   Rajani  7900.00        3
        Computer Science
3       40    Rushi  7623.45        1
        IT
4       50     Rutu  5823.41        3
        Computer Science
5       60  Prithvi  5399.14        3
```

```
Computer Science
```

- Link - pandas.DataFrame.merge — pandas 2.2.1 documentation

4.2 Pandas - Read and Write Files

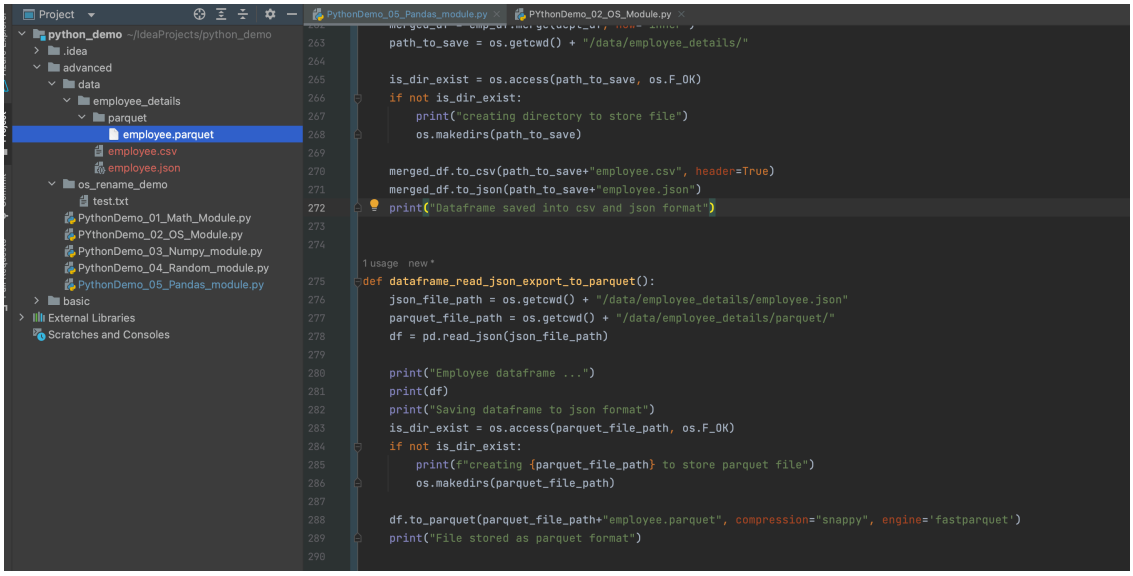
#Python/Advanced/Pandas

- We can use functions like read_csv, read_josn etc. to read csv, json files using pandas datframe
- Similarly to export any data frame to any format we can use functions like to_csv, to_json etc.

- **Export files using pandas data frame -**
  - to_csv(file_path+file_name.csv, header=true, sep='l')
    - by default csv separator ','
  - to_json(file_path+file_name.json)
  - to_parquet -
    - it requires engine either **fast parquet or pyarrow**
    - `pip install fastparquet`
  - e.g.

```
merged_df = emp_df.merge(dept_df, how='inner')
path_to_save = os.getcwd() + "/data/
    employee_details/"

is_dir_exist = os.access(path_to_save, os.F_OK)
if not is_dir_exist:
    print("creating directory to store file")
    os.makedirs(path_to_save)

merged_df.to_csv(path_to_save+"employee.csv",
    header=True)
```

```
merged_df.to_json(path_to_save+"employee.json")
print("Dataframe saved into csv and json
      format")
```



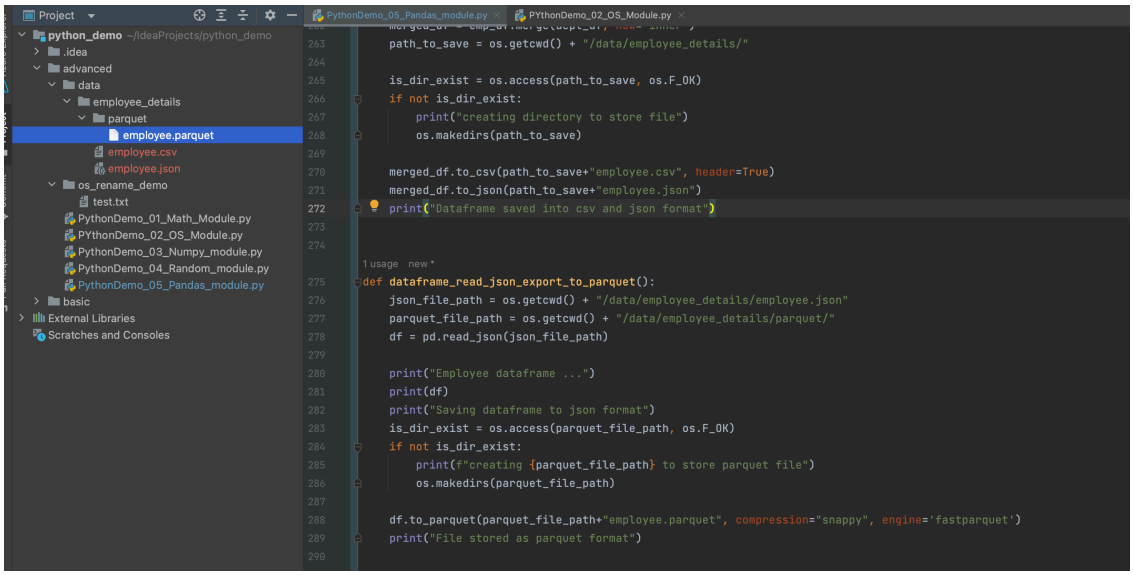
- **Read files using pandas data frame**
  - using functions like
    - pd.read_csv(file_path+file_name.csv)
    - pd.read_json(file_path+file_name.json)
    - pd.read_parquet(file_path+file_name.parquet)
  - e.g.

```
def dataframe_read_json_export_to_parquet():
    json_file_path = os.getcwd() + "/data/
     employee_details/employee.json"
    parquet_file_path = os.getcwd() + "/data/
     employee_details/parquet/"
    df = pd.read_json(json_file_path)

    print("Employee dataframe ...")
    print(df)
    print("Saving dataframe to json format")
    is_dir_exist = os.access(parquet_file_path,
     os.F_OK)
```

```
if not is_dir_exist:
    print(f"creating {parquet_file_path} to
 store parquet file")
    os.makedirs(parquet_file_path)


 df.to_parquet(parquet_file_path+"employee.
 parquet", compression="snappy",
 engine='fastparquet')
print("File stored as parquet format")
```

Output

```
  Employee dataframe ...
     emp_id emp_name  emp_sal  dept_id
dept_name
  0       10    Rohit  5600.00        1
IT
  1       20    Pooja  6200.00        2
Civil
  2       30   Rajani  7900.00        3  Computer
Science
  3       40    Rushi  7623.45        1
IT
  4       50     Rutu  5823.41        3  Computer
Science
  5       60  Prithvi  5399.14        3  Computer
Science
  Saving dataframe to json format
  File stored as parquet format
```


Project
python_demo ~/IdeaProjects/python_demo
.idea
advanced
data
employee_details
parquet
employee.parquet
employee.csv
employee.json
os_rename_demo
test.txt
PythonDemo_01_Math_Module.py
PYthonDemo_02_OS_Module.py
PythonDemo_03_Numpy_module.py
PythonDemo_04_Random_module.py
PythonDemo_05_Pandas_module.py
basic
External Libraries
Scratches and Consoles
PythonDemo_05_Pandas_module.py
PYthonDemo_02_OS_Module.py
path_to_save = os.getcwd() + "/data/employee_details/"
is_dir_exist = os.access(path_to_save, os.F_OK)
if not is_dir_exist:
print("creating directory to store file")
os.makedirs(path_to_save)
merged_df.to_csv(path_to_save+"employee.csv", header=True)
merged_df.to_json(path_to_save+"employee.json")
print("Dataframe saved into csv and json format")
1 usage new *
def dataframe_read_json_export_to_parquet():
json_file_path = os.getcwd() + "/data/employee_details/employee.json"
parquet_file_path = os.getcwd() + "/data/employee_details/parquet/"
df = pd.read_json(json_file_path)
print("Employee dataframe ...")
print(df)
print("Saving dataframe to json format")
is_dir_exist = os.access(parquet_file_path, os.F_OK)
if not is_dir_exist:
print(f"creating {parquet_file_path} to store parquet file")
os.makedirs(parquet_file_path)
df.to_parquet(parquet_file_path+"employee.parquet", compression="snappy", engine='fastparquet')
print("File stored as parquet format")