



Shri Vile Parle Kelavani Mandal's

**DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING**

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)



**A.Y. 2021-2022**

**Name:** Meith Navlakha

**SAP ID:** 60004190068

**Branch:** Computer Engineering

**Subject:** Python Lab Work

**Academic Year:** 2021-2022

**Semester:** V

**Division:** B

**Batch:** B1

## EXPERIMENT 1

**AIM:** Exploring basics of python like data types (strings, list, array, set, tuple) and control statements

### THEORY:

#### 1) Python Numbers:

Numeric data type represent the data which has numeric value. Numeric value can be integer, floating number or even complex numbers. These values are defined as int, float and complex class in Python.

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal). In Python there is no limit to how long an integer value can be.
- **Float** – This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- **Complex Numbers** – Complex number is represented by complex class. It is specified as  $(real\ part) + (imaginary\ part)j$ . For example –  $2+3j$

`type()` : function to know which class a variable or a value belongs to.

`isinstance()` function is used to check if an object belongs to a particular class.

#### 2) Python List

- An ordered sequence of items is called List.
- It is a very flexible data type in Python. It supports heterogeneous values i.e. it need not have only same data types.
- Lists are ordered, it means that the items have a defined order.If a new item is added into the list then it would be appended at the end.
- Allows duplicates as all the values have a unique index.
- Lists are mutable i.e. it can be updated.

### 3) Python Tuple

- Tuples are ordered collection used to store multiple items in a single variable.
- Tuples are initialised within parentheses (round brackets) with items being separated by commas.

Eg: Tuple1 = ('This', 'is', 1, 2+5j, 'Tuple')

- Creation of Python tuple without the use of parentheses is known as Tuple Packing.
- Tuples differ from list as tuples are immutable i.e. it cannot be modified.
- Tuples are used to write-protect data and are usually faster than lists as they cannot change dynamically.

### 4) Python String

- In Python, Strings are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote.  
Eg. 'hello' is the same as "hello".
- In python there is no character data type, a character is a string of length one. It is represented by str class.
- Strings are immutable
- Multi-line strings can be denoted using triple quotes, "" or """".
- Every individual characters of a String can be accessed by using the method of Indexing. Indexing even allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character and so on.

### 5) Python Set

- Set is an unordered collection of unique items. It arranges the
- Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.
- Eg: set1 = set([1, 2, Hello, 4.0, 'For', 6, Hello])

Output: {1, 2, Hello, 4.0, For, 6}

- Sets have unique values. They eliminate duplicates.

- We can perform set operations like union, intersection on two sets.
- Set items cannot be accessed by referring to an index, since sets are unordered the items has no index. We loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

## 6) Python Dictionary

- Dictionary is an unordered collection of data values, used to store data values like a map.
- Dictionary holds key:value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon : and each key is separated by a 'comma'.
- Dictionary keys are case sensitive, same name but different cases of Key will be treated distinctly.
- A dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated.
- Dictionary can also be created by the built-in function dict().

Eg: Dict = {1: 'Meith' , 2: 30 , 3: 'Hello'}

Dict = {'Name': 'Meith' , 'Age': 20 , 'Marks': [1, 2, 3, 4]}

- Dictionaries are mutable and data can be added ,updated or deleted.
- In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets. There is also a method called get() that will also help in accessing the element from a dictionary.

## 7) Python Arrays

- An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.
- Array can be handled in Python by a module named array. They can be useful when we have to manipulate only a specific data type values.
- A user can even treat lists as arrays. However, user cannot constraint the type of elements stored in a list. If you create arrays using the array module, all elements of the array must be of the same type.

- Array in Python can be created by importing array module.  
**array(data\_type, value\_list)** is used to create an array with data type and value list specified in its arguments.

## 8) Loops

Python programming language provides following types of loops to handle looping requirements. Python provides three ways for executing the loops. A). A loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string) While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

- A) For
- B) While
- C) For else
- D) Nested Loops

## 9) Control Statements

In general, statements are executed sequentially. There may be a situation when you need to execute a block of code several number of times. Control statements change execution from its normal sequence. Programming languages provide various control structures that allow for more complicated execution paths.

Python supports the following control statements:

- a) Break : Terminates the loop statement and transfers execution to the statement immediately following the loop.
- b) Continue : Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- c) Pass : The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

## CODE :

### 1) Python Numbers

```
m = 10
print (m , 'type of m : ' , type(m))

n = 5.0
print (n , 'type of n : ' , type(n))

k = 1 + 10j
print (k , 'type of k : ' , type(k))
print (k , 'is k complex? ' , isinstance(k , complex))
```

### OUTPUT:

```
10 type of m : <class 'int'>
5.0 type of n : <class 'float'>
(1+10j) type of k : <class 'complex'>
(1+10j) is k complex? True
```

### 2) Python List

```
# List
a= [10 , 2.4 , 'Hello' , 1+23j , 8]

# Operations on List
print('a[2] : ' , a[2]) # Fetching Single index
print('First 3 elements : ' , a[:3]) # slicing
print(a)

# List are Mutable
a[2] = 'Bye'
print('Updated Element : ' , a[2])
```

### OUTPUT:

```
a[2] : Hello
First 3 elements : [10, 2.4, 'Hello']
[10, 2.4, 'Hello', (1+23j), 8]
Updated Element : Bye
```

### 3) Python Tuple

# Initialise Tuple

```
a= (10 , 2.4 , 'Hello' , 1+23j , 8)
```

```
print(a)
```

```
print('a[2] : ' , a[2]) # Fetching by index
```

```
print('First 3 elements : ' , a[:3]) # Slicing tuple
```

#### OUTPUT:

```
(10, 2.4, 'Hello', (1+23j), 8)
```

```
a[2] : Hello
```

```
First 3 elements : (10, 2.4, 'Hello')
```

### 3) Python String

```
s = "I am learning python"
```

```
print(s)
```

```
print("s[5] : " , s[5])
```

```
print("s[6:11] : " , s[6:11])
```

# Multi Line String

```
l="A multiline
```

```
string"
```

```
print('\n' , l)
```

```
print("s[6:11] : " , s[6:11])
```

#### OUTPUT:

```
I am learning python
```

```
s[5] : l
```

```
s[6:11] : earni
```

```
A multiline
```

```
string
```

```
s[6:11] : earni
```

### 4) Python Set

# Python Set

```
a={4, 6, 1 , 8 ,9, 0 , 6 ,6}
```

```
print("a =" , a)
```

```
print(type(a))
```

## OUTPUT:

```
a = {0, 1, 4, 6, 8, 9}
```

```
<class 'set'>
```

## 5) Python Dictionary

```
d = {3:'val','meith':2}
print(type(d))
print("d[3] = ", d[3])
print("d['meith'] = ", d['meith'])
```

## OUTPUT:

```
<class 'dict'>
d[3] = val
d['meith'] = 2
```

## 6) Python Arrays

```
import array as arr
```

```
# Array with Integers
a = arr.array('i' , [1,2,3])
```

```
print('Integer Array1 :', a )
```

```
print('Elements of Array1 :', end=" ")
for i in range(len(a)):
    print(a[i], end=" ")
```

```
print()
```

```
# Accessing by Index
print('Index = 2 :', a[2] )
print('Negative Index= -2 :', a[-2] )
```

```
# Index Slicing
print("\nIndex Slicing : ' , a[1:3])
print('Index Slicing : ' , a[2:])
print('Negative Index Slicing : ' , a[-3:-1])
print()
```

```
b = arr.array('i' , [1 ,2, 5 ,6 , 8])
print("Initial Array : " , b)
```



### **Insert , Append , Pop and Delete in Array**

```
# Insert()
b.insert(1,4)
print('Updated Array after Insertion : ', b)

# Append()
b.append(11)
print('Updated Array after Append : ', b)

# POP
b.pop() # Removes element from the end.
print('Updated Array after POP : ', b)

b.pop(2) # Removes the 2nd last element
print('Updated Array after POP of 2nd Last Index : ', b)

# Delete
del b[3]
print('Updated Array after Deletion of index=3: ', b)
```

### **Arrays Update**

```
b = arr.array('i', [1 ,2, 5 ,6 , 8])

print("Array before updation : ", b)
b[2] = 10
print("Array after updation : ", b)

# Updating in loop
for i in range (len(b)):
    b[i] = b[i]*i
print("Array after updation : ", b)
```

### **Array Concatenation**

```
a = arr.array('d',[1.1 , 2.1 ,3.1,2.6,7.8])
b = arr.array('d',[3.7,8.6])
c = arr.array('d')
c=a+b
print("Array c = ",c)
```

### **OUTPUT:**

```
Integer Array1 : array('i', [1, 2, 3])
Elements of Array1 : 1 2 3
Index = 2 : 3
```

Negative Index= -2 : 2

Index Slicing : array('i', [2, 3])

Index Slicing : array('i', [3])

Negative Index Slicing : array('i', [1, 2])

### **Insert , Append , Pop and Delete in Array**

Initial Array : array('i', [1, 2, 5, 6, 8])

Updated Array after Insertion : array('i', [1, 4, 2, 5, 6, 8])

Updated Array after Append : array('i', [1, 4, 2, 5, 6, 8, 11])

Updated Array after POP : array('i', [1, 4, 2, 5, 6, 8])

Updated Array after POP of 2nd Last Index : array('i', [1, 4, 5, 6, 8])

Updated Array after Deletion of index=3: array('i', [1, 4, 5, 8])

### **Array Update**

Array before updation : array('i', [1, 2, 5, 6, 8])

Array after updation : array('i', [1, 2, 10, 6, 8])

Array after updation : array('i', [0, 2, 20, 18, 32])

### **Array Concatenation**

Array c = array('d', [1.1, 2.1, 3.1, 2.6, 7.8, 3.7, 8.6])

## **7) Python Conversions**

```
# Int to Float  
print(float(5))
```

```
# Float to Int  
print(int(10.5))  
print(int(-5.2))
```

```
# Float to String  
print(str(5.2))
```

```
# String to List  
print(list('hello'))
```

### **OUTPUT:**

```
5.0  
10  
-5  
5.2  
['h', 'e', 'l', 'l', 'o']
```

## 8) Loops

### A) For Loop

```
# Arrays
print("\nLooping through Array: ")
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)

# String
print("\nLooping through String: ")
for x in "banana":
    print(x)

print("\nLooping with break")
for x in fruits:
    print(x)
    if x == "banana":
        break
```

### OUTPUT:

Looping through Array:  
apple  
banana  
cherry

Looping through String:  
b  
a  
n  
a  
n  
a

Looping with break  
apple  
banana

### B)Nested For

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

**OUTPUT:**

red apple  
red banana  
red cherry  
big apple  
big banana  
big cherry  
tasty apple  
tasty banana  
tasty cherry

**B) For Else**

```
# for else
for x in range(6):
    print(x)
else:
    print("Finally finished!")
```

**OUTPUT:**

0  
1  
2  
3  
4  
5  
Finally finished!

**C) While**

```
count = 0
while (count < 3):
    count = count + 1
    print(count)
```

```
count = 0
print("\n Printing Even Numbers in Loop: ")
while (count < 10):
    if count%2 ==0:
        print(count)

    count = count + 1
```

## OUTPUT:

1  
2  
3

Printing Even Numbers in Loop:

0  
2  
4  
6  
8

## 9) Control Statements

a) break

```
for letter in 'thisismycode':  
    if letter == 'i' or letter == 's':  
        break  
    print('Current Letter :', letter)
```

b) continue

```
for letter in 'thisismycode':  
    if letter == 'i' or letter == 's':  
        continue  
    print('Current Letter :', letter)
```

c) pass

```
for letter in 'thisismycode':  
    if letter == 'i' or letter == 's':  
        pass  
    print('Current Letter :', letter)
```

## OUTPUT

### a) Break

```
# Break
for letter in 'thisismycode':
    if letter == 'e' or letter == 's':
        break
    print(letter)
```

t  
h  
i

### b) continue

```
for letter in 'thisismycode':
    if letter == 'i' or letter == 's':
        continue
    print(letter)
```

t  
h  
m  
y  
c  
o  
d  
e

### c) pass

```
# Pass
# Break
for letter in 'thisismycode':
    if letter == 'e' or letter == 's':
        pass
    print(letter)
```

t  
h  
i  
s  
i  
s  
m  
y  
c  
o  
d  
e

## **CONCLUSION:**

In this experiment we learnt about various data types in python programming. Python numbers which store numeric values – integers, float and complex. Python tuples and list which store a collection of data. Both are ordered sequence. The only difference between them is that tuples are immutable whereas lists are mutable. Python Sets which are a collection of unordered items. It only stores unique values i.e. does not duplicate entries. Python dictionary is an unordered collection of data values which stores the data in key-value pair. Python String is array of bytes representing Unicode characters. In python we have normal string and multiline string. Strings are immutable. Python arrays help in storing many data together in continuous memory location and it helps in storing the data together and access it easily. The data can be accessed easily by Indexing and are mutable. In loops we can execute a set of instructions repeatedly without repeating the code. Loops also have condition and other control statements to give us more control over the execution. Finally, we have Control Statements in python which are used to change the flow of execution for a certain condition. Python has 3 control statements viz, break, pass and continue.

## EXPERIMENT 2

**AIM:** To create functions, classes and objects using python

### **THEORY:**

Python is an object-oriented programming language. An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object. The methods of the object are basically the functions written in the class.

### **Functions in Python**

A function is a group of related statements that performs a specific task. Python Functions is a block of related statements designed to perform a computational, logical, or evaluative task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes the code reusable.

Functions can be both **built-in** or **user-defined**.

In Python a function is defined using the “def ” keyword.

The functions are defined and then called upon. To call a function, use the function name followed by parenthesis

### **Syntax of Function**

```
def function_name(parameters):
```

```
    """ docstring """  
    statement(s)
```

Function definition consists of the following components:

- Keyword ‘def ’ that marks the start of the function header.
- A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
- A colon (:) to mark the end of the function header.
- One or more valid python statements that make up the function body. Statements must have the same indentation level.
- Parameters (arguments) through which we pass values to a function. They are optional.
- Optional documentation string (docstring) to describe what the function does.
- An optional return statement to return a value from the function.



## Arguments in Functions

Any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

### Arbitrary Arguments (\*args)

If the number of arguments that will be passed into your function is undecided then add a \* before the parameter name in the function definition. This way the function will receive a tuple of arguments which can be accessed using index.

The default name is args, but we can use any other name e.g. \*names.

### Arbitrary Keyword Arguments (\*\*kwargs)

If the number of arguments that will be passed into your function is undecided then add a \* before the parameter name in the function definition. This way the function will receive a *dictionary* of arguments, and can access the items by using keys of the values.

## Recursion Function

Python also accepts function recursion, which means a defined function can call itself. Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

A complicated function can be split down into smaller sub-problems utilizing recursion. Sequence creation is simpler through recursion than utilizing any nested iteration. Recursive functions render the code look simple and effective.

## Lambda Functions in Python

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required. Lambda functions are used when we require a nameless function for a short period of time. Hence, are known as **Anonymous Functions** as well.

**Use:** In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

## Syntax

lambda arguments: expression

## Inner Functions in Python

A function which is defined inside another function is known as inner function or nested function. Nested functions are able to access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function. This process is also known as Encapsulation.

## Built-In Functions in Python

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

**abs(x)** : Return the absolute value of a number. The argument may be an integer, a floating point number, or an object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned.

**bin(x)** : Convert an integer number to a binary string prefixed with “0b”. The result is a valid Python expression. If x is not a Python int object, it has to define an `__index__()` method that returns an integer.

**chr(i)** : Return the string representing a character whose Unicode code point is the integer i. For example, `chr(97)` returns the string 'a', while `chr(8364)` returns the string '€'. The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). `ValueError` will be raised if i is outside that range.

**all(iterable)** : Return True if all elements of the iterable are true (or if the iterable is empty).

**len(s)** : Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

Built-in Functions			
<b>A</b> abs() aiter() all() any() anext() ascii()  <b>B</b> bin() bool() breakpoint() bytearray() bytes()  <b>C</b> callable() chr() classmethod() compile() complex()  <b>D</b> delattr() dict() dir() divmod()	<b>E</b> enumerate() eval() exec()  <b>F</b> filter() float() format() frozenset()  <b>G</b> getattr() globals()  <b>H</b> hasattr() hash() help() hex()  <b>I</b> id() input() int() isinstance() issubclass() iter()	<b>L</b> len() list() locals()  <b>M</b> map() max() memoryview() min()  <b>N</b> next()  <b>O</b> object() oct() open() ord()  <b>P</b> pow() print() property()	<b>R</b> range() repr() reversed() round()  <b>S</b> set() setattr() slice() sorted() staticmethod() str() sum() super()  <b>T</b> tuple() type()  <b>V</b> vars()  <b>Z</b> zip()  <b>_</b> __import__()

## Class in Python:

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

In Python, Class is defined by “def” keyword.

Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute. The values of the attributes of the objects of the class can also be modified.

## Objects in Python:

An Object is an instance of a Class. When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behaviour of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behaviour:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

## Constructors in Python:

The `__init__` method is similar to constructors in C++ and Java. Constructors are used to initializing the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated.

Use of `__init__()` function is to assign values to object properties, or other operations that are necessary to do when the object is being created.

## “self ” in Python:

Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it. This is similar to this pointer in C++ and this reference in Java.

The self parameter is a reference to the current instance of the class and is used to access variables that belongs to that class. Even If we have a method that takes no arguments, then we still have to have one argument.

## 1) Simple Function

```
# Function Defination
def my_func():
    print("Hello this is Meith")

# call function
my_func()
```

Hello this is Meith

## 2) Passing Arguments

```
# Function Definition
def calcArea(l):
    a= l*l
    print("Area of Square : " ,a)

# call function
calcArea(5)
```

Area of Square : 25

```
[35] def func(list):
      for x in list:
          print(x)

      func(['Apple' , 'Banana' , 'Mango' , 'Orange'])
```

Apple  
Banana  
Mango  
Orange

```
# Passsing arguements

def my_func2(name, age):
    print("Hello this is ", name)
    print("I am ", age , "years old.")
|
# call function
my_func2('Meith' , 20)
```

```
Hello this is  Meith
I am  20 years old.
```

---

### 3) Arbitrary Arguments

```
def smallestNum(*args):

    a =list(args)
    a.sort()
    print('Smallest Number : ' ,a[0])

print('The Numbers : 10, 5, 8, 20 , 6' )
smallestNum(10, 5, 8, 20 , 6)
```

```
The Numbers : 10, 5, 8, 20 , 6
Smallest Number :  5
```

### 4) Keyword Arguments

```
def my_function(child3, child2, child1):
    print("The chosen child is " + child3)

my_function(child1 = "Tom", child2 = "Tobias", child3 = "Meith")
```

```
The chosen child is Meith
```

## 5) Return

```
def isNum(num):  
    if (num>0):  
        return 'Positive'  
    elif (num<0):  
        return 'Negative'  
    else:  
        return 'Neither Positive nor Negative'  
  
# Calling Function  
print( '15 is :', isNum(15))  
print( '-8 is :', isNum(-8))  
print( ' 0 is :', isNum(0))
```

```
15 is : Positive  
-8 is : Negative  
 0 is : Neither Positive nor Negative
```

## 6) Scope of Variable

```
def my_func():  
    x = 10  
    print("Value inside function:",x)  
  
x = 20  
my_func()  
print("Value outside function:",x)
```

```
Value inside function: 10  
Value outside function: 20
```

## 7) Recursion

```
def fact(n):  
    if (n>0):  
        return (n*fact(n-1))  
    elif (n == 0):  
        return 1  
    else:  
        return 'Enter positive number!!'  
  
print("Factorial 5 :", fact(5))  
print("Factorial 0 :", fact(0))  
print("Factorial -4 :", fact(-4))
```

```
Factorial 5 : 120  
Factorial 0 : 1  
Factorial -4 : Enter positive number!!
```

## 8) Lambda Functions

```
multiply = lambda x: x*2  
multiply(5)
```

```
10
```

```
# Lambda function with filter
```

```
num = [10, 4, 5, 27, 67, 18, 36]  
print("All the numbers : " , num)  
  
divByThree = list(filter(lambda x: (x%3 == 0) , num))  
print("Divisible by 3: " , divByThree)
```

```
All the numbers : [10, 4, 5, 27, 67, 18, 36]  
Divisible by 3: [27, 18, 36]
```

---



## 9) Inner/Nested Functions

```
def outerfunc(s):  
  
    # define inner function  
    def innerfunc():  
        print('Hello,', s)  
  
    innerfunc() # call inner function  
  
# call outer function  
outerfunc('Meith')
```

Hello, Meith

### Scope of Variables in Nested Function

```
# Scope of Variables in Nested Functions  
def outerfunc():  
    s = 'In Outer Function'  
    # define inner function  
    def innerfunc():  
        s = 'In Inner Function'  
        print(s)  
  
    print(s)  
    innerfunc() # call inner function  
  
# call outer function  
outerfunc()
```

In Outer Function  
In Inner Function

## 10) Simple Class in Python

```
class Meith:
    m = 5

# instantiate object of class
myclass = Meith()
print("Print m variable of object myclass : " , myclass.m)

# Modify the values of object
myclass.m = 10
print("Print UPDATED m variable of object myclass : " , myclass.m)

# Add attribute
myclass.msg = "Hello"
|
print('Added a new attribute to the object : ' , myclass.msg)
```

```
Print m variable of object myclass : 5
Print UPDATED m variable of object myclass : 10
Added a new attribute to the object : Hello
```

## 10) Constructor in Python

```
class Students:

    field = 'Engineering'
    def __init__(self , name, age, course):

        self.name = name
        self.age = age
        self.course= course

# Initialise Student objects
Student1 = Students('Meith' , 20, 'Comps')
Student2 = Students('Python' , 19, 'IT')

# Print object values
print('*****STUDENT DETAILS*****')
print(Student1.name," : ", Student1.field , " " , Student1.course , " " , Student1.age)
print(Student2.name," : ", Student2.field , " " , Student2.course , " " , Student2.age)

*****STUDENT DETAILS*****
Meith : Engineering  Comps  20
Python : Engineering  IT   19
```

## 11) Object Method:

```
class Students:

    # field = 'Engineering'
    def __init__(self, name, marks, course):

        self.name = name
        self.marks = marks
        self.course = course

    def calcPass(self):
        # Method
        if(self.marks >= 35):
            return "Pass"
        else:
            return 'Fail'

# Initialise Student objects
Student1 = Students('Meith', 88, 'Comps')
Student2 = Students('Python', 19, 'IT')

# Print object values
print('*****STUDENT RESULTS*****\n')
print(Student1.name, " : ", Student1.course, " ", Student1.calcPass()) # calling calcPass method of the object
print(Student2.name, " : ", Student2.course, " ", Student2.calcPass())

*****STUDENT RESULTS*****

Meith : Comps Pass
Python : IT Fail
```

## CONCLUSION:

In this experiment we learnt about function, class and object in python. A function is a block of related statements that performs a specific task. In python we have inbuilt and user defined functions. In Built-In functions I explored abs, bin, len, all, chr functions. Also, functions can have parameters which take in arguments when the functions are called. When the number of arguments is not known, multiple arguments can also be passed using \*args which stores the arguments into a tuple and multiple key value pair as \*\*kwargs which stores the arguments into a dictionary. Lastly, in functions I also learnt about Inner/Nested functions, Recursion and Lambda functions in python. Class is the blueprint and object are instance of the class. Each class can have multiple objects, each have a unique pointer to it. Through objects we can access the variables and the methods of the class. Also, we can modify the values of the attributes of the object. In python, \_\_init\_\_ method is like a constructor and is called by default when an object is instantiated. Thus, these are the class and object that I learnt in the experiment.

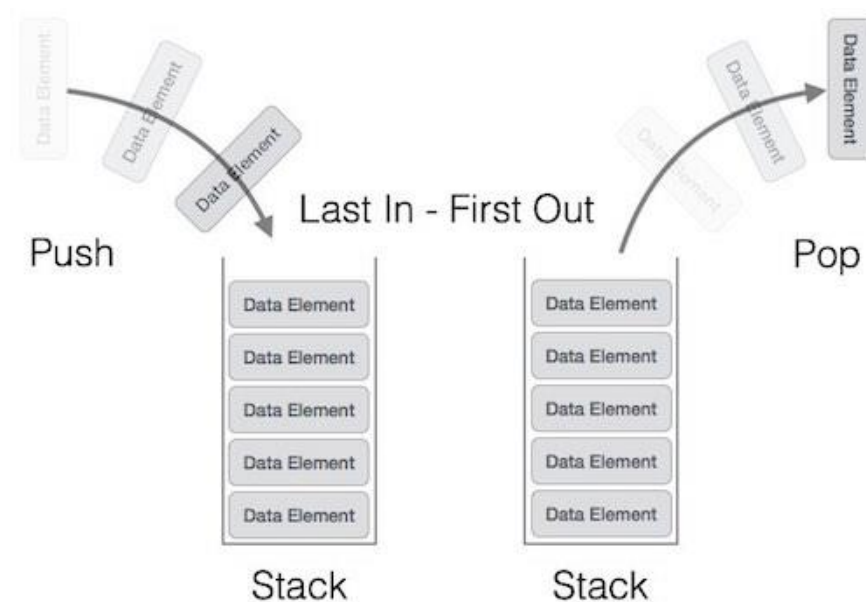
## EXPERIMENT 3

**AIM:** To implement data structures in Python.

**THEORY:**

**STACKS:**

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).



There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO (Last In First Out) / FILO (First In Last Out) order.

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations-

- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

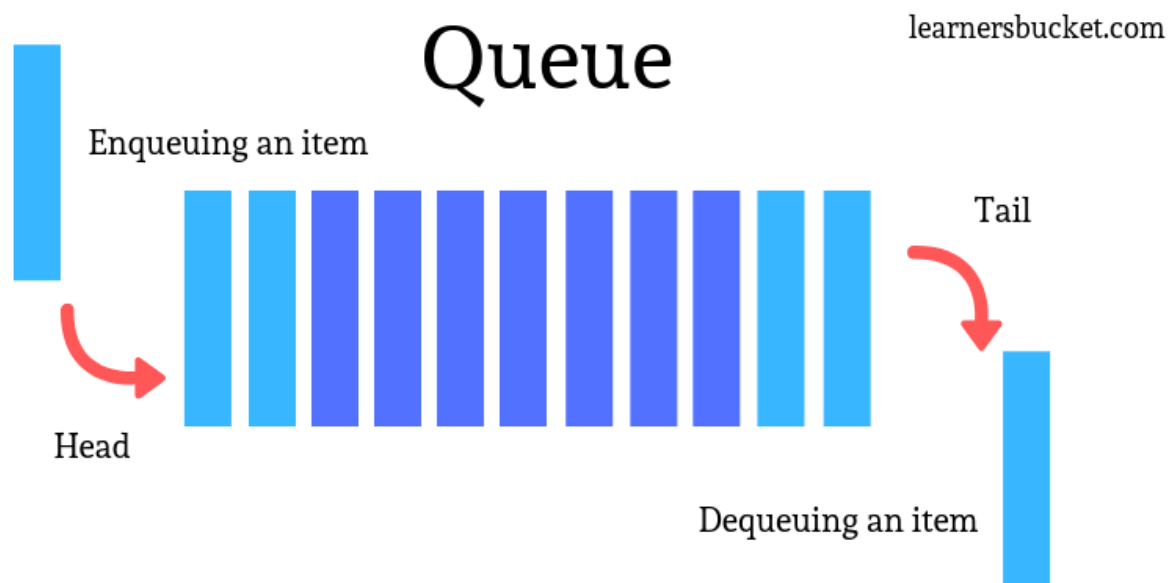
To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- **peek()** – get the top data element of the stack, without removing it.
- **isEmpty()** – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

## QUEUE:

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

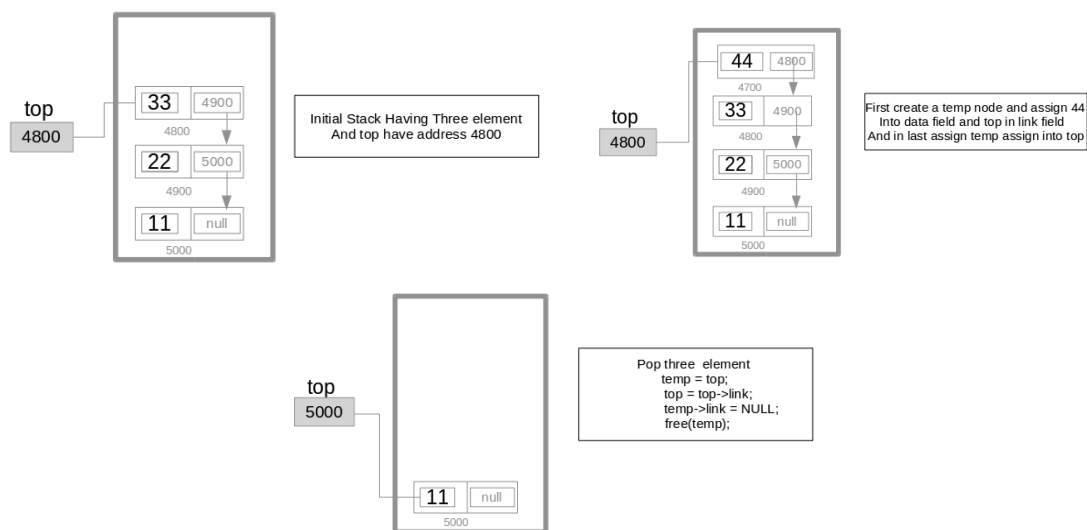
Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueueing (or storing) data in the queue we take help of **rear** pointer.

## Linked List as Stack:

Implement a stack using single linked list concept. all the single linked list operations perform based on Stack operations LIFO (last in first out) and with the help of that knowledge we are going to implement a stack using single linked list. using single linked lists so how to implement here it is linked list means what we are storing the information in the form of nodes and we need to follow the stack rules and we need to implement using single linked list nodes so what are the rules we need to follow in the implementation of a stack a simple rule that is last in first out and all the operations we should perform so with the help of a top variable only with the help of top variables are how to insert the elements let's see.



A stack can be easily implemented through the linked list. In stack Implementation, a stack contains a top pointer. which is “head” of the stack where pushing and popping items happens at the head of the list. first node has null in link field and second node link have first node address in link field and so on and last node address in “top” pointer. The main advantage of using linked list over an array is that it is possible to implements a stack that can shrink or grow as much as needed. In using array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated so overflow is not possible.

## Linked List as Queues:

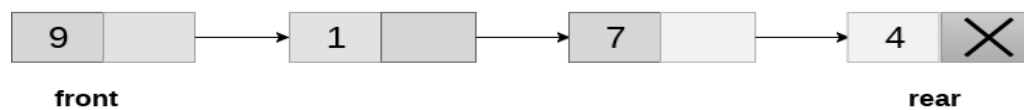
Due to the drawbacks discussed in the previous section of this tutorial, the array implementation cannot be used for the large scale applications where the queues are implemented. One of the alternatives of array implementation is linked list implementation of queue.

The storage requirement of linked representation of a queue with  $n$  elements is  $O(n)$  while the time requirement for operations is  $O(1)$ .

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue. Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



**Linked Queue**

## CODE:

### 1) STACK:

```
class Stack:
    # Initialise an empty stack.
    def __init__(self):
        self.data = []

    # PUSH
    def push(self, e):
        self.data.append(e)

    # POP
    def pop(self):
        if self.is_empty():
            # raise IndexError('Stack is empty')
            return 'STACK is Empty!!'
        else:
            return self.data.pop()

    # PEEK
    def peek(self):
        if self.is_empty():
            # raise IndexError('Stack is empty')
            return 'STACK is Empty!!'
        else:
            return self.data[-1]

    # isEmpty
    def is_empty(self):
        return len(self.data) == 0

    # SIZE
    def size(self):
        return len(self.data)
```

```
S = Stack()

# PUSH
S.push("S")
S.push("T")
print('Stack after PUSH: ', S.data)
S.push("A")
S.push("C")
S.push("K")
print('Stack after PUSH: ', S.data)

# PEEK
print('PEEK Stack : ', S.peek())
print('Stack Size : ', S.size())
print('Is Stack Empty ? ', S.is_empty())

# POP
S.pop()
S.pop()
S.pop()
S.pop()
S.pop()
print('Is Stack Empty after POP ? ', S.is_empty())
print('Stack size after POP ? ', S.size())
```

## OUTPUT:

```
Stack after PUSH:  ['S', 'T']
Stack after PUSH:  ['S', 'T', 'A', 'C', 'K']
PEEK Stack :  K
Stack Size :  5
Is Stack Empty ?  False
Is Stack Empty after POP ?  True
Stack size after POP ?  0
```



## 2) QUEUE:

```
# Queue Implementation

class QueueADT:
    # Initialise an empty queue.
    def __init__(self):
        self.data = []

    # ENQUEUE
    def enqueue(self, e):
        self.data.insert(0, e)

    # DEQUEUE
    def dequeue(self):
        if self.is_empty():
            # raise IndexError('Queue is empty')
            return 'QUEUE is Empty!!'
        else:
            return self.data.pop()

    # PEEK
    def peek(self):
        if self.is_empty():
            # raise IndexError('Queue is empty')
            return 'QUEUE is Empty!!'
        else:
            return self.data[-1]

    # isEmpty
    def is_empty(self):
        return len(self.data) == 0

    # SIZE
    def size(self):
        return len(self.data)

# Initialise Queue
Q = QueueADT()

# ADD
print('Is Queue Empty ? ', Q.is_empty() )
Q.enqueue("M")
Q.enqueue("E")
Q.enqueue("I")
Q.enqueue("T")
Q.enqueue("H")
print('Queue after ENQUEUE :', Q.data)
print('PEEK Queue : ', Q.peek())
print('Queue SIZE : ', Q.size() )
print('Is Queue Empty ? ', Q.is_empty() )

# DEQUEUE
Q.dequeue()
Q.dequeue()
Q.dequeue()
Q.dequeue()
Q.dequeue()
print('PEEK Queue after DEQUEUE :', Q.peek())
print('Queue SIZE after DEQUEUE : ', Q.size() )
print('Is Queue Empty ? ', Q.is_empty())
```

## OUTPUT:

```
Is Queue Empty ? True
Queue after ENQUEUE : ['H', 'T', 'I', 'E', 'M']
PEEK Queue : M
Queue SIZE : 5
Is Queue Empty ? False
PEEK Queue after DEQUEUE : QUEUE is Empty!!
Queue SIZE after DEQUEUE : 0
Is Queue Empty ? True
```

## 3) STACK Using Linked List:

```
class LinkedListStack:

    # NODE
    class Node:
        def __init__(self, e):
            self.element = e
            self.next = None

    # Initialise
    def __init__(self):
        self._size = 0
        # Reference --> top of stack
        self.head = None
        self.start = None

    # PUSH
    def push(self, e):

        newest = self.Node(e)
        if (self._size == 0 and self.start == None):
            self.start = newest
        newest.next = self.head
        self.head = newest
        self._size += 1

    # POP
    def pop(self):
        if self.is_empty():
            return 'Stack is Empty!!'
            # raise IndexError('Stack is empty')

        elementToReturn = self.head.element
        self.head = self.head.next
        self._size -= 1

        return elementToReturn
```

```
# PEEK
def peek(self):
    if self.is_empty():
        return 'Stack is Empty!!'
        # raise IndexError('Stack is empty')

    return self.head.element

# isEmpty.
def is_empty(self):
    return self._size == 0

# SIZE
def size(self):
    return self._size

def display(self):
    temp = self.head
    while (temp != None):
        print(temp.element, " ", end=" ")
        temp = temp.next

# Initialise LL
LLS = LinkedListStack()
print('Is STACK Empty ? ' , LLS.is_empty())

print('\n****AFTER PUSH****')
# PUSH
LLS.push("H")
LLS.push("T")
LLS.push("I")
LLS.push("E")
LLS.push("M")
print('Display STACK(TOP to Bottom) : ' ,end=" ")
LLS.display()
print('\nIs STACK Empty ? ' , LLS.is_empty())
print('PEEK Stack : ' , LLS.peek() )
print('Stack SIZE : ' , LLS.size() )
print('Is STACK Empty ? ' , LLS.is_empty())

# POP
LLS.pop()
LLS.pop()
LLS.pop()
LLS.pop()
LLS.pop()

print('\n****AFTER POP****')
print('POP on Empty STACK : ',LLS.pop() )
print('PEEK Stack : ' , LLS.peek() )
print('Stack SIZE : ' , LLS.size() )
print('Is STACK Empty ? ' , LLS.is_empty())
```

## OUTPUT:

```
Is STACK Empty ? True

****AFTER PUSH****
Display STACK(TOP to Bottom) : M E I T H
Is STACK Empty ? False
PEEK Stack : M
Stack SIZE : 5
Is STACK Empty ? False

****AFTER POP****
POP on Empty STACK : Stack is Empty!!
PEEK Stack : Stack is Empty!!
Stack SIZE : 0
Is STACK Empty ? True
```

## 4) QUEUE using Linked List:

```
class LinkedListQueue:

    class Node:
        def __init__(self, e):
            self.element = e
            self.next = None

    # Initialise
    def __init__(self):
        self._size = 0
        self.head = None
        self.tail = None

    # ADD
    def enqueue(self, e):
        newest = self.Node(e)

        if self.is_empty():
            self.head = newest
        else:
            self.tail.next = newest
            self.tail = newest
        self._size += 1

    # REMOVE
    def dequeue(self):
        if self.is_empty():
            return 'QUEUE is Empty!!'
            # raise IndexError('Queue is empty')

        elementToReturn = self.head.element
        self.head = self.head.next
        self._size -= 1
        if self.is_empty():
            self.tail = None

        return elementToReturn
```

```
# PEEK
def peek(self):
    if self.is_empty():
        return 'QUEUE is Empty!!'
        # raise IndexError('Queue is empty')
    return self.head.element

# isEmpty.
def is_empty(self):
    return self._size == 0

def size(self):
    return self._size
```

```
LLQ = LinkedListQueue()

# ADD
print('Is Queue Empty ? ' , Q.is_empty() )
LLQ.enqueue("M")
LLQ.enqueue("E")
LLQ.enqueue("I")
LLQ.enqueue("T")
LLQ.enqueue("H")
print('\n****AFTER ENQUEUE****')
print('PEEK Queue : ' , LLQ.peak())
print('Queue SIZE : ' , LLQ.size() )
print('Is Queue Empty ? ' , LLQ.is_empty() )

# DEQUEUE
LLQ.dequeue()
LLQ.dequeue()
LLQ.dequeue()
LLQ.dequeue()
LLQ.dequeue()
print('\n****AFTER DENQUEUE****')
print('PEEK Queue after DEQUEUE : ' , LLQ.peak())
print('Queue SIZE after DEQUEUE : ' , LLQ.size() )
print('Is Queue Empty ? ' , LLQ.is_empty())
```

## OUTPUT:

```
Is Queue Empty ?  True

****AFTER ENQUEUE****
PEEK Queue :  M
Queue SIZE :  5
Is Queue Empty ?  False

****AFTER DENQUEUE****
PEEK Queue after DEQUEUE : QUEUE is Empty!!
Queue SIZE after DEQUEUE :  0
Is Queue Empty ?  True
```

## 5) DEQUEUE

```
class Deque:

    def __init__(self):
        self.queue = []
        self.count = 0

    def display(self):

        if self.count == 0:
            print("Double Ended Queue Empty.")

        else:
            print("Double Ended Queue: " , end=" ")
            for i in range(self.count):
                print(" ",self.queue[i],end=" ")

    def insert_start(self, data):
        if self.count == 0:
            self.queue = [data,]
            self.count = 1
            return

        self.queue.insert(0, data)
        self.count += 1
        return

    def insert_end(self, data):
        if self.count == 0:
            self.queue = [data,]
            self.count = 1
            return

        self.queue.append(data)
        self.count += 1
        return
```

```
def remove_start(self):
    if self.count == 0:
        raise ValueError("Invalid Operation")

    x = self.queue.pop(0)
    self.count -= 1
    return x

def remove_end(self):
    if self.count == 0:
        raise ValueError("Invalid Operation")

    x = self.queue.pop()
    self.count -= 1
    return x

def get(self, index):
    if index >= self.count | index < 0:
        raise ValueError("Index out of range.")

    return self.queue[index]

def size(self):
    return self.count
```

```
# INITIALISE DEQUEUE OBJECT
d = Deque()

# EMPTY
print("\n\nNON INITIALISATION:")
d.display()

print("\n\nAFTER INSERTING AT START:")
# INSERT START
d.insert_start(10)
d.insert_start(8)
d.insert_start(74)
d.display()

print("\n\nAFTER INSERTING AT END:")
# INSERT END
d.insert_end(3)
d.insert_end(54)
d.insert_end(-8)
d.display()

# POPPING FRONT
print("\n\nAFTER POPING FROM START: ")
d.remove_start()
d.display()

# POPING FROM END
print("\n\nAFTER POPING FROM END: ")
d.remove_end()
d.display()

# SIZE
print("\n\nDQUEUE SIZE: ", d.size())
```

## OUTPUT:

```
ON INITIALISATION:
Double Ended Queue Empty.

AFTER INSERTING AT START:
Double Ended Queue:  74  8  10

AFTER INSERTING AT END:
Double Ended Queue:  74  8  10  3  54  -8

AFTER POPING FROM START:
Double Ended Queue:  8  10  3  54  -8

AFTER POPING FROM END:
Double Ended Queue:  8  10  3  54

DQUEUE SIZE:  4
```

## CONCLUSION:

In this experiment, I have implemented different data structure like Stacks, Queues, Stack using Linked List, Queue using Linked List and Dequeue in python language. I have made a class for each of the data structure along with their basic methods like push, pop, peek isEmpty for these data structures and later instantiate the object and using the methods defined to perform the necessary operations which can be seen in the screenshots of the outputs. Thus, I learnt how to code and implement the data structures in python from this experiment.



## EXPERIMENT 4

**AIM:** To implement exception handling in Python.

### THEORY:

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).

When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.

If never handled, an error message is displayed, and our program comes to a sudden unexpected halt. Errors occur at specific times. The two major time frames are

- Compile time
- Runtime

### Compile time error:

A compile time error occurs when you ask Python to run the application. Before Python can run the application, it must interpret the code and put it into a form that the computer can understand. A computer relies on machine code that is specific to that processor and architecture. If the instructions you write are malformed or lack needed information, Python can't perform the required conversion. It presents an error that you must fix before the application can run.

Fortunately, compile-time errors are the easiest to spot and fix. Because the application won't run with a compile-time error in place, user never sees this error category. You fix this sort of error as you write your code.

### Runtime error:

A runtime error occurs after Python compiles the code you write and the computer begins to execute it. Runtime errors come in several different types, and some are harder to find than others. You know you have a runtime error when the application suddenly stops running and displays an exception dialog box or when the user complains about erroneous output (or at least instability).

Many runtime errors are caused by errant code. For example, you can misspell the name of a variable, preventing Python from placing information in the correct variable during execution. Leaving out an optional but necessary argument when calling a method can also cause problems. These are examples of *errors of commission*, which are specific errors associated with your code. In general, you can find these kinds of errors by using a debugger or by simply reading your code line by line to check for errors.

Runtime errors can also be caused by external sources not associated with your code. For example, the user can input incorrect information that the application isn't expecting, causing an exception. A network error can make a required resource inaccessible. Sometimes even the computer hardware has a glitch that causes a nonrepeatable application error. These are all examples of *errors of omission*, from which the application might recover if your application has error-trapping code in place. It's important that you consider both kinds of runtime errors — errors of commission and omission — when building your application

The trick is to know where to look. With this in mind, Python (and most other programming languages) breaks errors into the following types:

- Syntactical
- Semantic
- Logical

## **Syntactical**

Whenever you make a typo of some sort, you create a syntactical error. Some Python syntactical errors are quite easy to find because the application simply doesn't run. The interpreter may even point out the error for you by highlighting the errant code and displaying an error message. However, some syntactical errors are quite hard to find. Python is case sensitive, so you may use the wrong case for a variable in one place and find that the variable isn't quite working as you thought it would. Finding the one place where you used the wrong capitalization can be quite challenging.

## **Semantic**

When you create a loop that executes one too many times, you don't generally receive any sort of error information from the application. The application will happily run because it thinks that it's doing everything correctly, but that one additional loop can cause all sorts of data errors. When you create an error of this sort in your code, it's called a *semantic error*.

Semantic errors occur because the meaning behind a series of steps used to perform a task is wrong — the result is incorrect even though the code apparently runs precisely as it should. Semantic errors are tough to find, and you sometimes need some sort of debugger to find them.

## **Logical**

Some developers don't create a division between semantic and logical errors, but they are different. A semantic error occurs when the code is essentially correct but the implementation is wrong (such as having a loop execute once too often). Logical errors occur when the developer's thinking is faulty. In many cases, this sort of error happens when the developer uses a relational or logical operator incorrectly. However, logical errors can happen in all sorts of other ways, too. For example, a developer might think

that data is always stored on the local hard drive, which means that the application may behave in an unusual manner when it attempts to load data from a network drive instead.

## Catching Exceptions in Python

The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented. Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks. The try: block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent except: block is skipped. If the exception does occur, the program flow is transferred to the except: block. The statements in the except: block are meant to handle the cause of the exception appropriately. For example, returning an appropriate error message. You can specify the type of exception after the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate. The rest of the statements after the except block will continue to be executed, regardless if the exception is encountered or not. In Python, keywords else and finally can also be used along with the try and except clauses. While the except block is executed if the exception occurs inside the try block, the else block gets processed if the try block is found to be exception free.

The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. Consequently, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code. If, however, there's an exception in the try block, the appropriate except block will be processed, and the statements in the finally block will be processed before proceeding to the rest of the code.

```
try:
    You do your operations here;
    .....
    Due to any exception, this may be skipped.
finally:
    This would always be executed.
    .....
```

## Raise an Exception

Python also provides the `raise` keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.

### The *assert* Statement

When it encounters an `assert` statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception. The **syntax** for `assert` is :

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses `ArgumentExpression` as the argument for the *AssertionError*. *AssertionError* exceptions can be caught and handled like any other exception using the `try-except` statement, but if not handled, they will terminate the program and produce a traceback.

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Some common exception classes in python:

Sr.No.	Exception Name & Description
1	<b>Exception</b> Base class for all exceptions
2	<b>StopIteration</b> Raised when the <code>next()</code> method of an iterator does not point to any object.
3	<b>SystemExit</b> Raised by the <code>sys.exit()</code> function.
4	<b>StandardError</b> Base class for all built-in exceptions except <code>StopIteration</code> and <code>SystemExit</code> .
5	<b>ArithmeticError</b> Base class for all errors that occur for numeric calculation.
6	<b>OverflowError</b> Raised when a calculation exceeds maximum limit for a numeric type.

7	<b>FloatingPointError</b> Raised when a floating point calculation fails.
8	<b>ZeroDivisionError</b> Raised when division or modulo by zero takes place for all numeric types.
9	<b>AssertionError</b> Raised in case of failure of the Assert statement.
10	<b>AttributeError</b> Raised in case of failure of attribute reference or assignment.
11	<b>EOFError</b> Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	<b>ImportError</b> Raised when an import statement fails.
13	<b>KeyboardInterrupt</b> Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	<b>LookupError</b> Base class for all lookup errors.
15	<b>IndexError</b> Raised when an index is not found in a sequence.
16	<b>KeyError</b> Raised when the specified key is not found in the dictionary.
17	<b>NameError</b> Raised when an identifier is not found in the local or global namespace.
18	<b>UnboundLocalError</b> Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	<b>EnvironmentError</b> Base class for all exceptions that occur outside the Python environment.
20	<b>IOError</b> Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.

## CODE:

### 1) IndexError

```
[1] a = [1, 2, 3, 4, 5]

try:
    print(a[5])

except IndexError:
    print("Index does not exist !")
```

Index does not exist !

### 2) assert

```
a = int(input())

assert a >= 100, "Number entered was not greater than 100 !"
print(a)
```

100  
100

### 3) Exception Handling

```
import sys

num = ['a', 0, 2]

for i in num:
    try:
        print("The entry is", i)
        r = 1/int(i)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occurred.\n")

print("The reciprocal of", i, "is", r)
```

```
The entry is a
Oops! <class 'ValueError'> occurred.

The entry is 0
Oops! <class 'ZeroDivisionError'> occurred.

The entry is 2
The reciprocal of 2 is 0.5
```

```
import sys

num = ['a', 0, 2]

for i in num:
    try:
        print("The entry is", i)
        r = 1/int(i)
        break
    except Exception as e:
        print("Oops!", e.__class__, "occurred.\n")


print("The reciprocal of", i, "is", r)
```

The entry is a  
Oops! <class 'ValueError'> occurred.

The entry is 0  
Oops! <class 'ZeroDivisionError'> occurred.

The entry is 2  
The reciprocal of 2 is 0.5

#### 4) Specific Error

 `raise KeyboardInterrupt`

```
-----
KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-9-c761920b81b0> in <module>()
----> 1 raise KeyboardInterrupt
```

KeyboardInterrupt:

[SEARCH STACK OVERFLOW](#)

[ ] `raise MemoryError("Raised Sample Memory Error")`

```
-----
MemoryError                                Traceback (most recent call last)
<ipython-input-11-f43800e550bb> in <module>()
----> 1 raise MemoryError("Raised Sample Memory Error")
```

MemoryError: Raised Sample Memory Error

[SEARCH STACK OVERFLOW](#)

#### 5) Catch Specific Error

```
[ ]
try:
    a = int(input("Enter a positive integer: "))
    if a <= 0:
        raise ValueError("Not a positive number!")
except ValueError as ve:
    print(ve)
```

Enter a positive integer: -5  
Not a positive number!

## 6) try catch with if

```
[7] try:
    a = int(input("Enter a positive integer: "))
    if a <= 0:
        raise ValueError("That is not a positive number!")
except ValueError as ve:
    print(ve)
```

Enter a positive integer: -8  
That is not a positive number!

```
[ ] # Reciprocal of Even
num = 0
while(num != -1):

    try:
        num = int(input("Enter an Even Number: "))
        if(num == -1 ):
            break
        assert num % 2 == 0
    except:
        print("Not an even number!")
    else:
        reciprocal = 1/num
        print("Reciprocal of num : ",reciprocal)
```

Enter an Even Number: 7  
Not an even number!  
Enter an Even Number: 14  
Reciprocal of num : 0.07142857142857142  
Enter an Even Number: -1  
Not an even number!

## 6) try with assert

```
[17] try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

Enter a number: 5  
Not an even number!



## 7) try except Ladder

```
[ ] arr = ['a' , 4, 0 , 9]

for elem in arr:

    try:
        print("\nEntry :" , elem)
        assert elem % 2 == 0
        rec = 1/int(elem)
        # break

    except ValueError:
        print("The number must be even!!")
        pass

    except ZeroDivisionError:
        print("Cannot find Reciprocal of 0!!")
        pass

    except TypeError:
        print("Cannot find Reciprocal of String")
        pass

    except:
        print(sys.exc_info()[0])
        pass

    else:
        print("Reciprocal of ", elem," : ", rec )
```

Entry : a  
Cannot find Reciprocal of String

Entry : 4  
Reciprocal of 4 : 0.25

Entry : 0  
Cannot find Reciprocal of 0!!

Entry : 9  
<class 'AssertionError'>

## 8) try except finally

```
[21] try:
    print('try block')
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x=0
    y=0
print ("Out of try, except, else and finally blocks." )
```

```
try block
Enter a number: 5
Enter another number: 0
except ZeroDivisionError block
Division by 0 not accepted
finally block
Out of try, except, else and finally blocks.
```

## 9) Custom Error

```
[28] class Networkerror(RuntimeError):
    def __init__(self, arg):
        self.args = arg

    try:
        raise Networkerror("Bad hostname")
    except Networkerror as e:
        print("".join(e.args))
```

```
Bad hostname
```

## **CONCLUSION:**

In this experiment, I learnt about the two types of errors namely, runtime error and compile time error. Python has built-in exception handling in case of an error. The error can be caught using a try-except block and appropriate code can be written in the except block to handle the error. In this experiment, I have implemented try – except, try-except ladder(handling multiple exception), try-except-finally. I have also, developed by custom error and raised it in the code. Also in the experiment I have learnt about assert keyword that helps to put certain constraint or restriction on a particular variable or expression.

## EXPERIMENT 5

**AIM:** Python program to explore different types of Modules

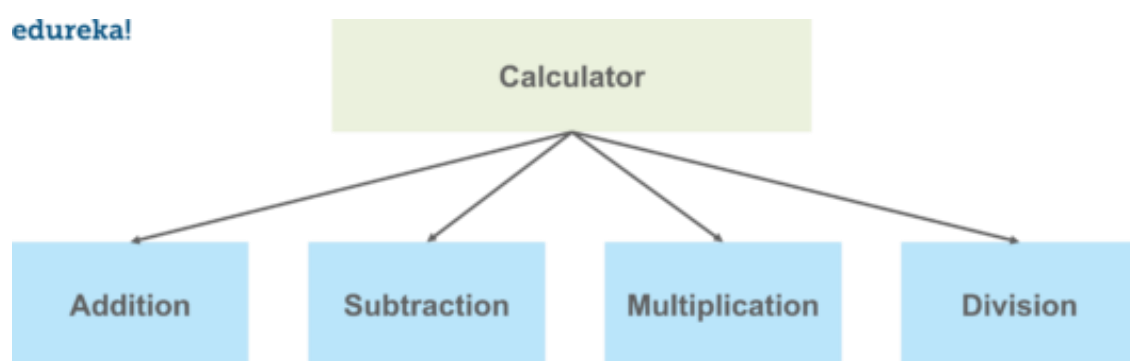
### THEORY:

In Python, **Modules** are simply files with the “.py” extension containing Python code that can be imported inside another Python Program. A module can be considered as a code library or a file that contains a set of functions that you want to include in your application. With the help of modules, we can organize related functions, classes, or any code block in the same file.

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules. The module contains the following components:

- Definitions and implementation of classes,
- Variables, and
- Functions that can be used inside another program.

Suppose we want to make an application for a calculator. We want to include few operations in our application such as addition, subtraction, multiplication, division, etc. Now, we will break the code into separate parts and simply create one module for all these operations or separate modules for each of the operations. And then we can call these modules in our main program logic.



### Types of Models:

1. A module can be written in Python itself.
2. A module can be written in C and loaded dynamically at run-time, like the re (regular expression) module.
3. A built-in module is intrinsically contained in the interpreter, like the itertools module.

## Create and Access Python Modules

To create a module, we have to save the code in a file with the file extension “.py”. Then, the name of the Python file becomes the name of the module. The module’s contents are accessed with the import statement.

### Advantages of Modules

- Reusability: Working with modules makes the code reusable.
- Simplicity: The module focuses on a small proportion of the problem, rather than focusing on the entire problem.
- Scoping: A separate namespace is defined by a module that helps to avoid collisions between identifiers.

### CODE:

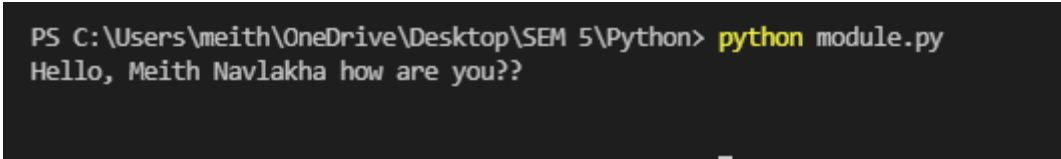
#### welcome.py - (MODULE)

```
def welcome(name):  
    print("Hello, " + name + " how are you??")
```

#### module.py - (MAIN FILE)

```
from welcome import welcome  
welcome("Meith Navlakha")
```

### OUTPUT:



```
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python module.py  
Hello, Meith Navlakha how are you??
```

### CODE:

#### person.py (MODULE)

```
person1 = {  
    "name": "Meith Navlakha",  
    "age": 21,  
    "city": "India",  
    "college": "D. J. Sanghvi"  
}
```

### Module.py (MAIN FILE):

```
import person  
a = person.person1["age"]  
b = person.person1["name"]  
c = person.person1["college"]
```

```
print("\nNAME : ", b)  
print("AGE : ",a)  
print("COLLEGE : ",c)
```

### OUTPUT:

```
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python module.py  
  
NAME :  Meith Navlakha  
AGE :  21  
COLLEGE :  D. J. Sanghvi
```

### Rename a Python Module:

We can name the file of the module whatever you like, but we have to note that it must have the file extension **“.py”**. To rename the module name, we can create an alias when you import a module, with the help of the **as keyword**

### CODE:

```
import person as mod  
a = mod.person1["age"]  
b = mod.person1["name"]  
c = mod.person1["college"]
```

```
print("\nNAME : ", b)  
print("AGE : ",a)  
print("COLLEGE : ",c)
```

### OUTPUT:

```
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python module.py  
  
NAME :  Meith Navlakha  
AGE :  21  
COLLEGE :  D. J. Sanghvi
```

## Python Built-in Modules

Python has a number of built-in functions. The methods are loaded automatically and are always available, such as,

- `print()` and `input()` for I/O,
- Number conversion functions such as `int()`, `float()`, `complex()`,
- Data type conversions such as `list()`, `tuple()`, `set()`, etc.

In addition to these many built-in functions, there are also a large number of pre-defined functions available as a part of libraries bundled with Python distributions. These functions are defined in modules which are known as **built-in modules**.

These built-in modules are written in C language and integrated with the Python shell. Most useful and frequently used built-in modules of Python.

- Math Module
- Statistics Module

## Math Module of Python

Some of the most popular mathematical functions that are defined in the math module include,

- Trigonometric functions,
- Representation functions,
- Logarithmic functions,
- Angle conversion functions, etc.

In addition, two mathematical constants- **pi** and **e** are also defined in this module.

## CODE:

```
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.radians(45)
0.7853981633974483
>>> math.degrees(math.pi/3)
59.99999999999999
>>> math.sin(0.785398)
0.7071066656470943
>>> math.cos(0.785398)
0.7071068967259817
>>> math.tan(0.785398)
0.9999996732051568
>>> math.sqrt(16)
4.0
>>> math.ceil(4.2)
5
>>> math.floor(4.7)
4
>>> math.pow(9,3)
729.0
>>> math.log(2)
0.6931471805599453
>>> math.log10(2)
0.3010299956639812
>>> |
```

## Statistics Module of Python

The statistics module provides functions to mathematical statistics of numeric data. Some of the popular statistical functions are defined in this module are as follows:

- Mean
- Median
- Mode
- Standard Deviation



## CODE

```
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import statistics
>>> statistics.mean([5,10,15])
10
>>> statistics.median([1,2,3,4,5,6])
3.5
>>> statistics.mode([2,2,2,5,5,6,4,4,5,5])
5
>>> statistics.stdev([1,1.5,2,2.5,3,3.5,4])
1.0801234497346435
>>> |
```

## Datetime Module of Python

In Python, date and time are not a data type of their own, but a module named **datetime** can be imported to work with the date as well as time. **Python Datetime module** comes built into Python, so there is no need to install it externally.

Python Datetime module supplies classes to work with date and time. These classes provide a number of functions to deal with dates, times and time intervals. Date and datetime are an object in Python, so when you manipulate them, you are actually manipulating objects and not string or timestamps.

The DateTime module is categorized into 6 main classes –

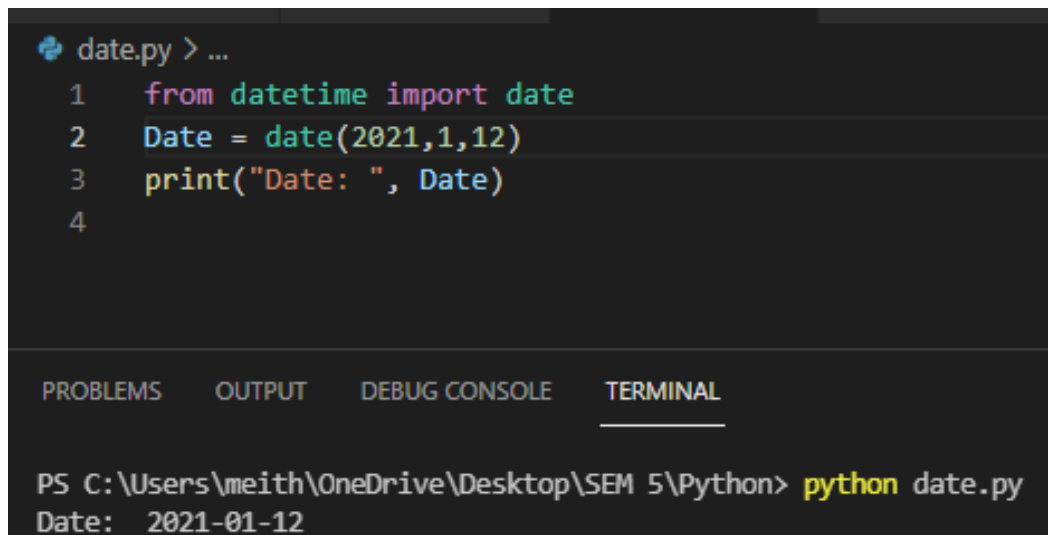
- **date** : An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Its attributes are year, month and day.
- **time** : An idealized time, independent of any particular day, assuming that every day has exactly 24\*60\*60 seconds. Its attributes are hour, minute, second, microsecond, and tzinfo.
- **datetime** : Its a combination of date and time along with the attributes year, month, day, hour, minute, second, microsecond, and tzinfo.
- **timedelta** : A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.
- **tzinfo** : It provides time zone information objects.
- **timezone** : A class that implements the tzinfo abstract base class as a fixed offset from the UTC

## CODE:

### 1) Basic

```
from datetime import date  
Date = date(2021,1,12)  
print("Date: ", Date)
```

## OUTPUT:

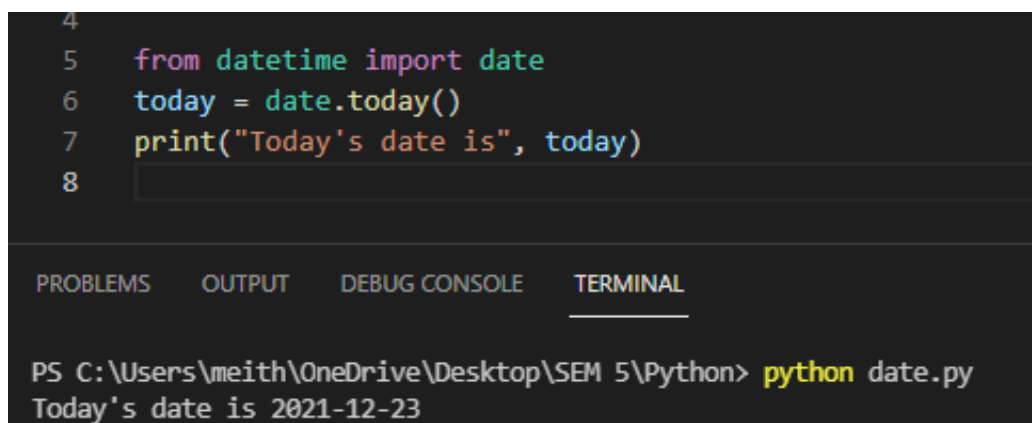


```
date.py > ...  
1  from datetime import date  
2  Date = date(2021,1,12)  
3  print("Date: ", Date)  
4  
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
  
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python date.py  
Date: 2021-01-12
```

### 2) today()

```
from datetime import date  
today = date.today()  
print("Today's date is", today)
```

## OUTPUT:



```
4  
5  from datetime import date  
6  today = date.today()  
7  print("Today's date is", today)  
8  
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
  
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python date.py  
Today's date is 2021-12-23
```

### 3) Month, Year, Day

```
from datetime import date
today = date.today()
print("Current year:", today.year)
print("Current month:", today.month)
print("Current day:", today.day)
```

#### OUTPUT:

```
10  from datetime import date
11  today = date.today()
12  print("Current year:", today.year)
13  print("Current month:", today.month)
14  print("Current day:", today.day)
15
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python date.py
Current year: 2021
Current month: 12
Current day: 23
```

### 4) Get date from Timestamp

```
from datetime import datetime
date_time = datetime.fromtimestamp(1887639468)
print("Datetime from timestamp:", date_time)
```

#### OUTPUT:

```
16  from datetime import datetime
17  date_time = datetime.fromtimestamp(1887639708)
18  print("Datetime from timestamp:", date_time)
19
20
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python date.py
Datetime from timestamp: 2029-10-25 21:51:48
```

**5)Convert Date to String :** Can convert date object to a string representation using two functions isoformat() and strftime().

```
from datetime import date
today = date.today()
Str = date.isoformat(today)
print("String Representation", Str)
print(type(Str))
```

**OUTPUT:**

```
20
21  from datetime import date
22  today = date.today()
23  Str = date.isoformat(today)
24  print("String Representation", Str)
25  print(type(Str))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python date.py
String Representation 2021-12-23
<class 'str'>
```

**6) Time object representing time in Python**

```
from datetime import time
my_time = time(13, 24, 56)
print("Entered time", my_time)
my_time = time(minute=12)
print("\nTime with one argument", my_time)
my_time = time()
print("\nTime without argument", my_time)
```

**OUTPUT:**

```
27  from datetime import time
28  my_time = time(10, 22, 45)
29  print("Entered time", my_time)
30  my_time = time(minute=25)
31  print("Time with one argument", my_time)
32  my_time = time()
33  print("Time without argument", my_time)
34
```

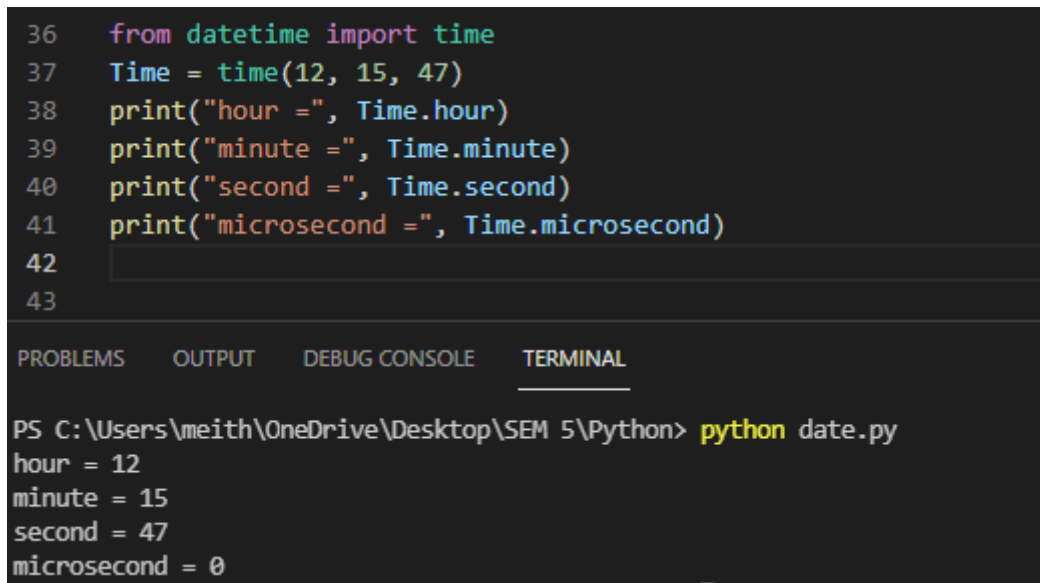
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python date.py
Entered time 10:22:45
Time with one argument 00:25:00
Time without argument 00:00:00
```

## 7) Get hours, minutes, seconds, and microseconds

```
from datetime import time
Time = time(12, 15, 47)
print("hour =", Time.hour)
print("minute =", Time.minute)
print("second =", Time.second)
print("microsecond =", Time.microsecond)
```

### OUTPUT:



The screenshot shows a Python IDE with a dark theme. The top part displays the code from the previous block, with line numbers 36 to 43. The bottom part shows the output of the code, which is: hour = 12, minute = 15, second = 47, and microsecond = 0. The IDE has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL, with the TERMINAL tab selected.

```
36 from datetime import time
37 Time = time(12, 15, 47)
38 print("hour =", Time.hour)
39 print("minute =", Time.minute)
40 print("second =", Time.second)
41 print("microsecond =", Time.microsecond)
42
43
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python date.py  
hour = 12  
minute = 15  
second = 47  
microsecond = 0

## 8)Add days to datetime object

```
from datetime import datetime, timedelta
ini_time_for_now = datetime.now()
print("initial_date", str(ini_time_for_now))
future_date_after_2yrs = ini_time_for_now + timedelta(days=730)

future_date_after_2days = ini_time_for_now + timedelta(days=2)

print('future_date_after_2yrs:', str(future_date_after_2yrs))
print('future_date_after_2days:', str(future_date_after_2days))
```

## OUTPUT:

```
43  from datetime import datetime, timedelta
44  ini_time_for_now = datetime.now()
45  print("initial_date", str(ini_time_for_now))
46  future_date_after_200d = ini_time_for_now + timedelta(days=200)
47
48  future_date_after_10d = ini_time_for_now + timedelta(days=10)
49  |
50  print('future_date_after_200d:', str(future_date_after_200d))
51  print('future_date_after_10d:', str(future_date_after_10d))
52
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python date.py
initial_date 2021-12-23 22:42:29.217255
future_date_after_200d: 2022-07-11 22:42:29.217255
future_date_after_10d: 2022-01-02 22:42:29.217255
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> █
```

## 8) Difference between two date and times

```
from datetime import datetime, timedelta
ini_time_for_now = datetime.now()
print("initial_date", str(ini_time_for_now))
new_final_time = ini_time_for_now + timedelta(days=5)
print("new_final_time", str(new_final_time))
print("Time difference:", str(new_final_time - ini_time_for_now))
```

## OUTPUT:

```
53  from datetime import datetime, timedelta
54  ini_time_for_now = datetime.now()
55  print("initial_date", str(ini_time_for_now))
56  new_final_time = ini_time_for_now + timedelta(days=5)
57  print("new_final_time", str(new_final_time))
58  print('Time difference:', str(new_final_time - ini_time_for_now))
59
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
PS C:\Users\meith\OneDrive\Desktop\SEM 5\Python> python date.py
initial_date 2021-12-23 22:51:36.204779
new_final_time 2021-12-28 22:51:36.204779
Time difference: 5 days, 0:00:00
```

**CONCLUSION:** In this experiment I have successfully studied and implemented different types of modules in python. Python has 3 types of modules: module in Python itself, module can be written in C and loaded dynamically and built-in module. I have created my module welcome.py and imported it in the module.py file. Also, in this experiment I have learnt and implemented some built-in modules like statistics, time and math.

## EXPERIMENT 6

**AIM:** Demonstrate File handling and Directories

1. Python program to append data to existing file and then display the entire file.
2. Python program to count number of lines, words and characters in a file.
3. Python program to display file available in current directory

### THEORY:

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).

Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

Open a file

Read or write (perform operation)

Close the file

Opening Files in Python

Python has a built-in `open()` function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt") # open file in current directory
```

```
>>> f = open("C:/Python38/README.txt") # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read `r`, write `w` or append `a` to the file. We can also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.



Mode	Description
r	Opens a file for reading. (default)
w	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Opens a file for exclusive creation. If the file already exists, the operation fails.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Opens in text mode. (default)
b	Opens in binary mode.
+	Opens a file for updating (reading and writing)

```
f = open("test.txt")    # equivalent to 'r' or 'rt'
```

```
f = open("test.txt",'w') # write in text mode
```

```
f = open("img.bmp",'r+b') # read and write in binary mode
```

Unlike other languages, the character a does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings).

Moreover, the default encoding is platform dependent. In windows, it is cp1252 but utf-8 in Linux.

So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt", mode='r', encoding='utf-8')
```

## **Closing Files in Python**

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file. It is done using the `close()` method available in Python.

Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.

```
f = open("test.txt", encoding = 'utf-8')  
  
# perform file operations  
  
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use an exception-handling block.

try:

```
f = open("test.txt", encoding = 'utf-8')  
  
# perform file operations
```

finally:

```
f.close()
```

This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop.

The best way to close a file is by using the `with` statement. This ensures that the file is closed when the block inside the `with` statement is exited.

We don't need to explicitly call the `close()` method. It is done internally.

with `open("test.txt", encoding = 'utf-8')` as `f`:

```
# perform file operations
```

## **Writing to Files in Python**

In order to write into a file in Python, we need to open it in write `w`, append `a` or exclusive creation `x` mode.

We need to be careful with the `w` mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using the `write()` method. This method returns the number of characters written to the file.

with open("test.txt",'w',encoding = 'utf-8') as f:

```
f.write("my first file\n")
```

```
f.write("This file\n\n")
```

```
f.write("contains three lines\n")
```

This program will create a new file named test.txt in the current directory if it does not exist. If it does exist, it is overwritten.

We must include the newline characters ourselves to distinguish the different lines.

## **Reading Files in Python**

To read a file in Python, we must open the file in reading r mode.

There are various methods available for this purpose. We can use the read(size) method to read in the size number of data. If the size parameter is not specified, it reads and returns up to the end of the file.

We can read the test.txt file we wrote in the above section in the following way:

```
>>> f = open("test.txt",'r',encoding = 'utf-8')
```

```
>>> f.read(4)  # read the first 4 data
```

```
'This'
```

```
>>> f.read(4)  # read the next 4 data
```

```
'is '
```

```
>>> f.read()   # read in the rest till end of file
```

```
'my first file\nThis file\ncontains three lines\n'
```

```
>>> f.read()  # further reading returns empty sting
```

```
''
```

We can see that the read() method returns a newline as '\n'. Once the end of the file is reached, we get an empty string on further reading.

We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).

```
>>> f.tell() # get the current file position
```

```
56
```

```
>>> f.seek(0) # bring file cursor to initial position
```

```
0
```

```
>>> print(f.read()) # read the entire file
```

```
This is my first file
```

```
This file
```

```
contains three lines
```

We can read a file line-by-line using a for loop. This is both efficient and fast.

```
>>> for line in f:
```

```
...     print(line, end = '')
```

```
...
```

```
This is my first file
```

```
This file
```

```
contains three lines
```

In this program, the lines in the file itself include a newline character `\n`. So, we use the `end` parameter of the `print()` function to avoid two newlines when printing.

Alternatively, we can use the `readline()` method to read individual lines of a file. This method reads a file till the newline, including the newline character.

```
>>> f.readline()
```

```
'This is my first file\n'
```

```
>>> f.readline()
```

```
'This file\n'
```

```
>>> f.readline()
'contains three lines\n'
```

```
>>> f.readline()
''
```

Lastly, the `readlines()` method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.

```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three lines\n']
```

## Python File Methods

There are various methods available with the file object. Some of them have been used in the above examples.

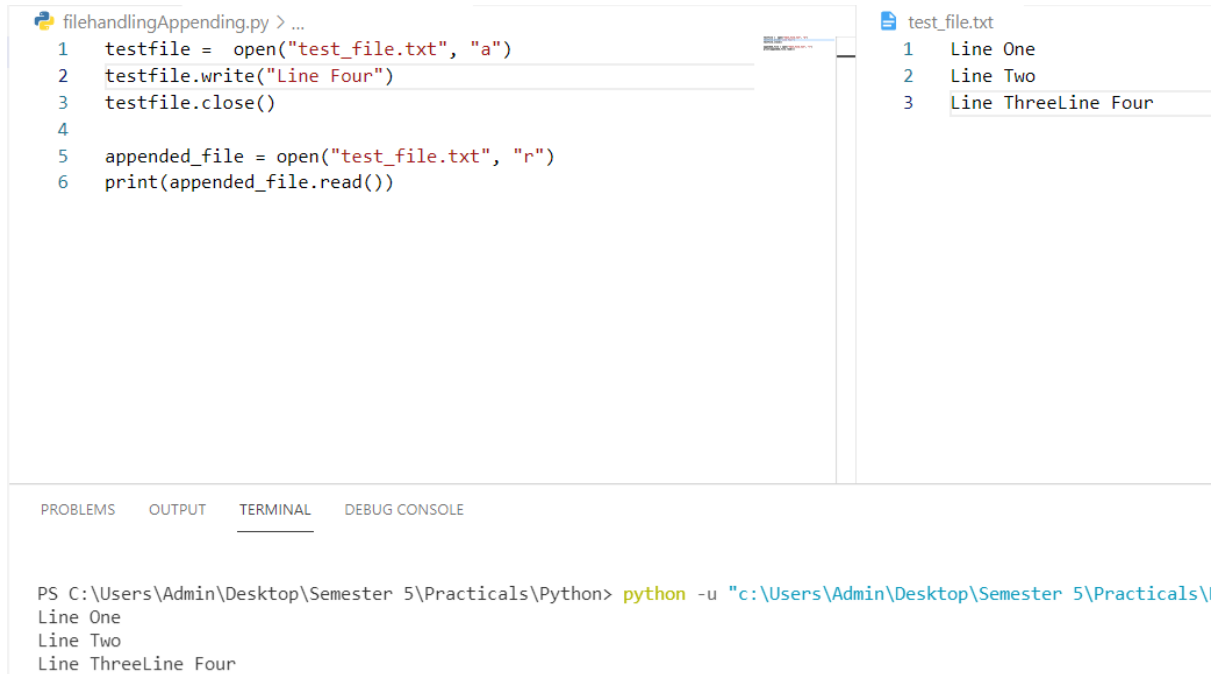
Here is the complete list of methods in text mode with a brief description:

Method	Description
<code>close()</code>	Closes an opened file. It has no effect if the file is already closed.
<code>detach()</code>	Separates the underlying binary buffer from the <code>TextIOBase</code> and returns it.
<code>fileno()</code>	Returns an integer number (file descriptor) of the file.
<code>flush()</code>	Flushes the write buffer of the file stream.
<code>isatty()</code>	Returns <code>True</code> if the file stream is interactive.
<code>read(n)</code>	Reads at most <code>n</code> characters from the file. Reads till end of file if it is negative or <code>None</code> .

<code>readable()</code>	Returns True if the file stream can be read from.
<code>readline(n=-1)</code>	Reads and returns one line from the file. Reads in at most n bytes if specified.
<code>readlines(n=-1)</code>	Reads and returns a list of lines from the file. Reads in at most n bytes/characters if specified.
<code>seek(offset,from=SEEK_SET)</code>	Changes the file position to offset bytes, in reference to from (start, current, end).
<code>seekable()</code>	Returns True if the file stream supports random access.
<code>tell()</code>	Returns the current file location.
<code>truncate(size= None)</code>	Resizes the file stream to size bytes. If size is not specified, resizes to current location.
<code>writable()</code>	Returns True if the file stream can be written to.
<code>write(s)</code>	Writes the string s to the file and returns the number of characters written.
<code>writelines(lines)</code>	Writes a list of lines to the file.

## CODE:

1) Python program to append data to existing file and then display the entire file.



The screenshot shows a Python IDE with a file named `filehandlingAppending.py` and a file named `test_file.txt`. The code in the editor is as follows:

```
1 testfile = open("test_file.txt", "a")
2 testfile.write("Line Four")
3 testfile.close()
4
5 appended_file = open("test_file.txt", "r")
6 print(appended_file.read())
```

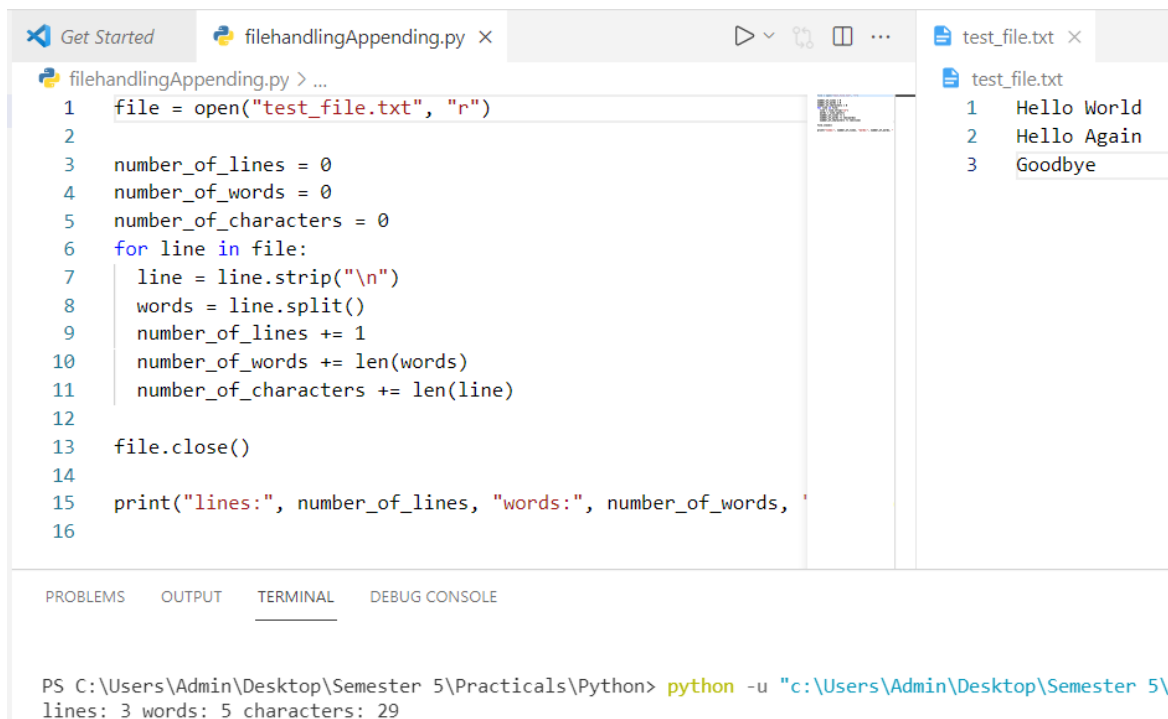
The `test_file.txt` file contains the following text:

```
1 Line One
2 Line Two
3 Line ThreeLine Four
```

The terminal output shows the execution of the program:

```
PS C:\Users\Admin\Desktop\Semester 5\Practicals\Python> python -u "c:\Users\Admin\Desktop\Semester 5\Practicals\
Line One
Line Two
Line ThreeLine Four
```

2) Python program to count number of lines, words and characters in a file.



The screenshot shows a Python IDE with a file named `filehandlingAppending.py` and a file named `test_file.txt`. The code in the editor is as follows:

```
1 file = open("test_file.txt", "r")
2
3 number_of_lines = 0
4 number_of_words = 0
5 number_of_characters = 0
6 for line in file:
7     line = line.strip("\n")
8     words = line.split()
9     number_of_lines += 1
10    number_of_words += len(words)
11    number_of_characters += len(line)
12
13 file.close()
14
15 print("lines:", number_of_lines, "words:", number_of_words, '
16
```

The `test_file.txt` file contains the following text:

```
1 Hello World
2 Hello Again
3 Goodbye
```

The terminal output shows the execution of the program:

```
PS C:\Users\Admin\Desktop\Semester 5\Practicals\Python> python -u "c:\Users\Admin\Desktop\Semester 5\
lines: 3 words: 5 characters: 29
```

### 3) Python program to display file available in current directory

```
C: > filehandlingAppending.py > ...  
1 import os  
2 arr = os.listdir('.')  
3 print("Files in the current directory are: ")  
4 for i in arr:  
5     print(i)
```

PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

```
PS C:\> python .\filehandlingAppending.py  
Files in the current directory are:  
$Recycle.Bin  
$WINDOWS.BT  
$Windows.WS  
$WinREAgent  
AVScanner.ini  
Documents and Settings  
DumpStack.log  
DumpStack.log.tmp  
filehandlingAppending.py  
hiberfil.sys  
Intel  
MSI  
mylog.log  
pagefile.sys  
PerfLogs  
Program Files  
Program Files (x86)  
ProgramData  
Recovery  
RHDSetup.log  
swapfile.sys  
System Volume Information  
Users  
Windows
```

### CONCLUSION:

In this experiment, I have implemented file handling mechanism and have also explored certain functionalities of the OS module in python programming language. Using Python we can append data to existing file, read data from the file, count the number of lines, words or characters in a file and even display the files available in current directory.



## EXPERIMENT 7

**AIM:** Make use of RE module to perform text processing.

### **THEORY:**

Regular expressions, also called regex is implemented in pretty much every computer language. In python, it is implemented in the standard module `re`. It is widely used in natural language processing, web applications that require validating string input (like email address) and pretty much most data science projects that involve text mining.

### **Using Regular Expressions in Python**

To start using Regex in Python, you first need to import Python's "`re`" module

#### **`re.findall()`: Finding all matches in a string/list**

Regex's `findall()` function is extremely useful as it returns a list of strings containing all matches. If the pattern is not found, `re.findall()` returns an empty list.

The `findall()` function takes two parameters, the first is the pattern being searched and the second parameter is text we are searching through.

With text data, there may be times where you need to extract words that are followed by a special character such as `@` for usernames or quotes within the given text.

We set our pattern to `"(.+?)"`, where the single quotes represent the text body, while the double quotes represent the quotes within the text. We have the parenthesis which creates a capture group and `.+?` is a non-greedy moderator and only extracts the quotes and not the text between the quotes.

#### **`re.match()`: Returning first occurrence in text**

While `re.findall()` returns matches of a substring found in a text, `re.match()` searches only from the beginning of a string and returns match object if found. However, if a match is found somewhere in the middle of the string, it returns none.

The expression `"w+"` and `"\W"` will match the words starting with letter 'r' and thereafter, anything which is not started with 'r' is not identified.

#### **`re.search()`: Finding pattern in text**

The `re.search()` function will search the regular expression pattern and return the first occurrence. Unlike Python `re.match()`, it will check all lines of the input string. If the pattern is found, the match object will be returned, otherwise "null" is returned.

## CODE:

### a) Regex Module

```
import re
txt = "Use of python in Machine Learning"
x = re.search("^Use.*Learning$", txt)
if (x):
    print("Match Found")
else:
    print("No match")
```

## OUTPUT:

```
PS D:\sem 5\python\exp7> python regex1.py
Match Found
PS D:\sem 5\python\exp7> █
```

### b) findall

```
import re
txt = "Use of python in Machine Learning"
x = re.findall("in", txt)
print(x)
```

## OUTPUT:

```
PS D:\sem 5\python\exp7> python regex2.py
['in', 'in', 'in']
PS D:\sem 5\python\exp7> █
```

### c) search

```
import re
txt = "Hello World"
searchObj = re.search("\s", txt)
print("The first white-space character is located at position: ", searchObj.start())
```

## OUTPUT:

```
PS D:\sem 5\python\exp7> python regex3.py
The first white-space character is located at position: 5
PS D:\sem 5\python\exp7> █
```

**d) split**

```
import re
string = "Python is one of the most popular languages around the world"
searchObj = re.split("\s", string)
print(searchObj)
```

**OUTPUT:**

```
PS D:\sem 5\python\exp7> python regex4.py
['Python', 'is', 'one', 'of', 'the', 'most', 'popular', 'languages', 'around', 'the', 'world']
PS D:\sem 5\python\exp7> █
```

**e) sub**

```
import re
string = "Python is one of the most popular language around the world"
searchObj = re.sub("\s", "_", string)
print(searchObj)
```

**OUTPUT:**

```
PS D:\sem 5\python\exp7> python regex5.py
Python_is_one_of_the_most_popular_language_around_the_world
PS D:\sem 5\python\exp7> █
```

**f) Match**

```
import re
string = "Python is one of the most popular language around the world"
searchObj = re.search("on", string)
print(searchObj)
```

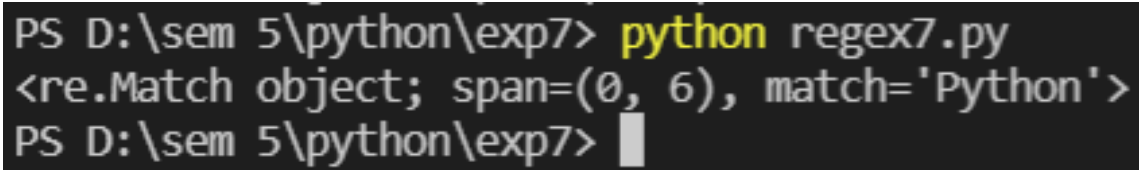
**OUTPUT:**

```
PS D:\sem 5\python\exp7> python regex6.py
<re.Match object; span=(4, 6), match='on'>
PS D:\sem 5\python\exp7> █
```

**g) Look for a specific character and case**

```
import re
string = "Python is one of the most popular language around the world"
searchObj = re.search(r"\bP\w+", string)
print(searchObj)
```

## OUTPUT:

A screenshot of a Windows command prompt window. The prompt shows the directory path 'D:\sem 5\python\exp7'. The user has entered the command 'python regex7.py'. The output of the script is displayed on the next line: '<re.Match object; span=(0, 6), match='Python'>'. The prompt is ready for the next command.

```
PS D:\sem 5\python\exp7> python regex7.py
<re.Match object; span=(0, 6), match='Python'>
PS D:\sem 5\python\exp7>
```

## CONCLUSION:

In this experiment, I have learnt about Regular Expression in python. In python, it is implemented using standard module 're'. In this experiment, I learnt and implemented various methods of re package like findall, search, match and sub. The screenshots of the same is attached above. RegEx is used form validation, routing path validation, text mining and in many other areas.

## EXPERIMENT 8

**AIM:** Creating GUI with python containing widgets such as labels, radio, checkbox and custom dialogue boxes.

### THEORY:

Python offers multiple options for developing GUI (Graphical User Interface). Out of all the GUI methods, tkinter is the most commonly used method. It is a standard Python interface to the Tk GUI toolkit shipped with Python. Python with tkinter is the fastest and easiest way to create the GUI applications. Creating a GUI using tkinter is an easy task.

#### To create a tkinter app:

1. Importing the module – tkinter
2. Create the main window (container)
3. Add any number of widgets to the main window
4. Apply the event Trigger on the widgets.

There are two main methods used which the user needs to remember while creating the Python application with GUI.

1. **Tk(screenName=None, baseName=None, className='Tk', useTk=1):** To create a main window, tkinter offers a method 'Tk(screenName=None, baseName=None, className='Tk', useTk=1)'. To change the name of the window, you can change the className to the desired one. The basic code used to create the main window of the application is:

m=tkinter.Tk() , where m is the name of the main window object

2. **mainloop():** There is a method known by the name mainloop() is used when your application is ready to run. mainloop() is an infinite loop used to run the application, wait for an event to occur and process the event as long as the window is not closed.

m.mainloop()

tkinter also offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows. There are mainly three geometry manager classes class.

1. **pack() method:**It organizes the widgets in blocks before placing in the parent widget.
2. **grid() method:**It organizes the widgets in grid (table-like structure) before placing in the parent widget.
3. **place() method:**It organizes the widgets by placing them on specific positions directed by the programmer.

There are a number of widgets which you can put in your tkinter application. Some of the major widgets are explained below:

1. **Button:** To add a button in your application, this widget is used.

The general syntax is:

`w=Button (master, option=value)`

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the Buttons. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **activebackground:** to set the background color when button is under the cursor.
- **activeforeground:** to set the foreground color when button is under the cursor.
- **bg:** to set the normal background color.
- **command:** to call a function.
- **font:** to set the font on the button label.
- **image:** to set the image on the button.
- **width:** to set the width of the button.
- **height:** to set the height of the button.

2. **Canvas:** It is used to draw pictures and other complex layout like graphics, text and widgets.

The general syntax is:

`w = Canvas (master, option=value)`

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bd:** to set the border width in pixels.
- **bg:** to set the normal background color.
- **cursor:** to set the cursor used in the canvas.
- **highlightcolor:** to set the color shown in the focus highlight.
- **width:** to set the width of the widget.
- **height:** to set the height of the widget.

3. **CheckBox:** To select any number of options by displaying a number of options to a user as toggle buttons. The general syntax is:

`w = CheckButton (master, option=value)`

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **Title:** To set the title of the widget.
- **activebackground:** to set the background color when widget is under the cursor.
- **activeforeground:** to set the foreground color when widget is under the cursor.
- **bg:** to set the normal background color.
- **secretcode:** to set the secret code.

Attach a File: nd color.

- **command:** to call a function.
- **font:** to set the font on the button label.
- **image:** to set the image on the widget.

4. **Entry:** It is used to input the single line text entry from the user. For multiline text input, Text widget is used

The general syntax is:

`w=Entry (master, option=value)`

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bd:** to set the border width in pixels.
- **bg:** to set the normal background color.
- **cursor:** to set the cursor used.
- **command:** to call a function.
- **highlightcolor:** to set the color shown in the focus highlight.
- **width:** to set the width of the button.
- **height:** to set the height of the button.

5. **Frame:** It acts as a container to hold the widgets. It is used for grouping and organizing the widgets. The general Syntax is :

`w = Frame(master, option=value)` ... master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
- **bd**: to set the border width in pixels.
- **bg**: to set the normal background color.
- **cursor**: to set the cursor used.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

6. **Label**: It refers to the display box where you can put any text or image which can be updated and time as per the code.

```
w=Label (master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bg**: to set he normal background color.
- **bg** to set he normal background color.
- **command**: to call a function.
- **font**: to set the font on the button label.
- **image**: to set the image on the button.
- **width**: to set the width of the button.
- **height**” to set the height of the button.

## CODE:

```
from tkinter import messagebox
from tkinter import *
root = Tk ()
root.title('Login Page')
root.geometry('500x280')
f= Frame(root,height=400,width=500)
f.pack()

l1=Label(f,text='Username',width=20,height=2)
l2=Label(f,text='Password',width=20,height=2)
l3=Label(f,text='Address',width=20,height=2)
l4=Label(f,text='Favoutite Subject',width=20,height=2)
l5=Label(f,text='Gender',width=20,height=2)
```



```
e1=Entry(f,width=20)
e2=Entry(f,width=20,show=('*'))
t= Text(f,width=20,height=5)
c1=Checkbutton(f,text='DWM')
c2=Checkbutton(f,text='AI')
c3=Checkbutton(f,text='Python')

def hellocallback():
    s1=e1.get()
    s2=e2.get()
    if s1=='test' and s2=='pass':
        messagebox.showinfo('Login Message', 'Welcome Login Successful...')
    else:
        messagebox.showerror('Login Message', 'Invalid User....')

var=IntVar()
r1=Radiobutton(f,text='Male',variable=var,value=1)
r2=Radiobutton(f,text='Female',variable=var,value=2)

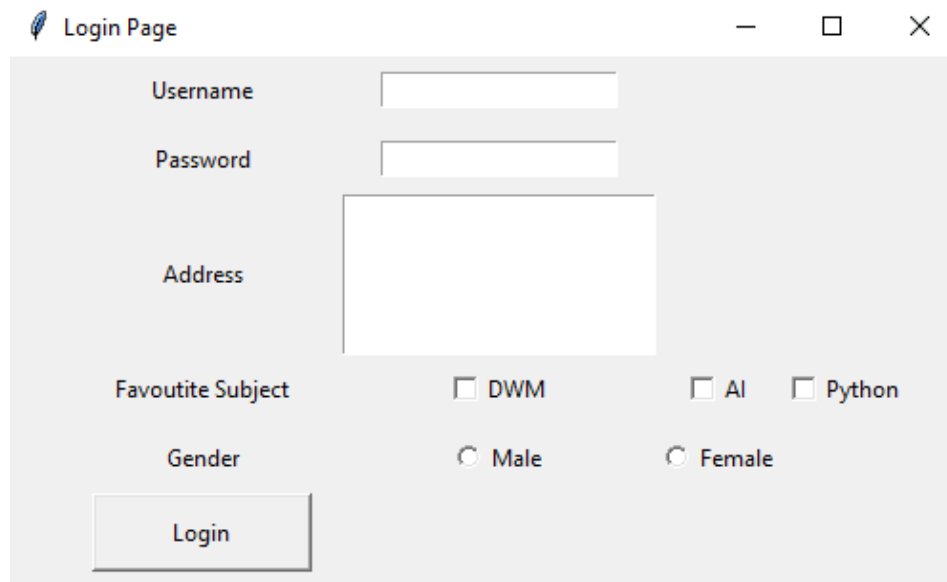
b=Button(f,text='Login',width=15,height=2,command=hellocallback)

#For alignment

l1.grid(row=0,column=0)
e1.grid(row=0,column=1)
l2.grid(row=1,column=0)
e2.grid(row=1,column=1)
l3.grid(row=2,column=0)
t.grid(row=2,column=1)
l4.grid(row=3,column=0)
c1.grid(row=3,column=1)
c2.grid(row=3,column=2)
c3.grid(row=3,column=3)
l5.grid(row=4,column=0)
r1.grid(row=4,column=1)
r2.grid(row=4,column=2)
b.grid(row=5,column=0)
root.mainloop()
```

## OUTPUT:

Before any input:

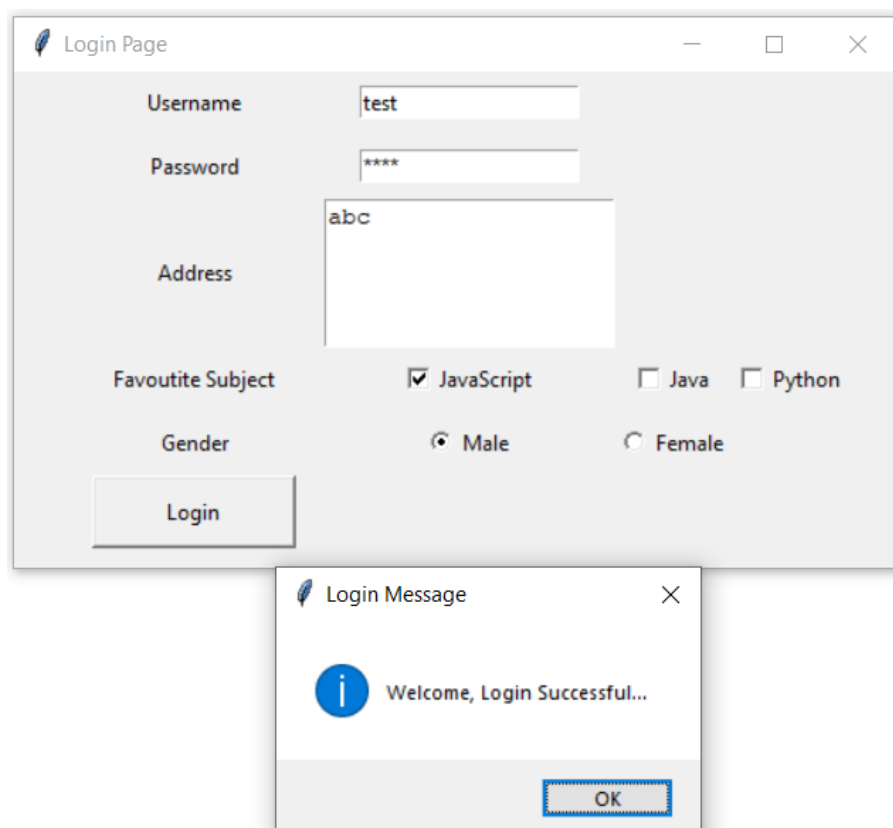


A screenshot of a Windows application window titled "Login Page". The window contains a form with the following fields and controls:

- Username:** A text input field.
- Password:** A text input field.
- Address:** A larger text input field.
- Favourite Subject:** A group of three checkboxes: ☐ DWM, ☐ AI, and ☐ Python.
- Gender:** A group of two radio buttons: ☐ Male and ☐ Female.
- Login:** A button at the bottom left.

After Input:

If valid input



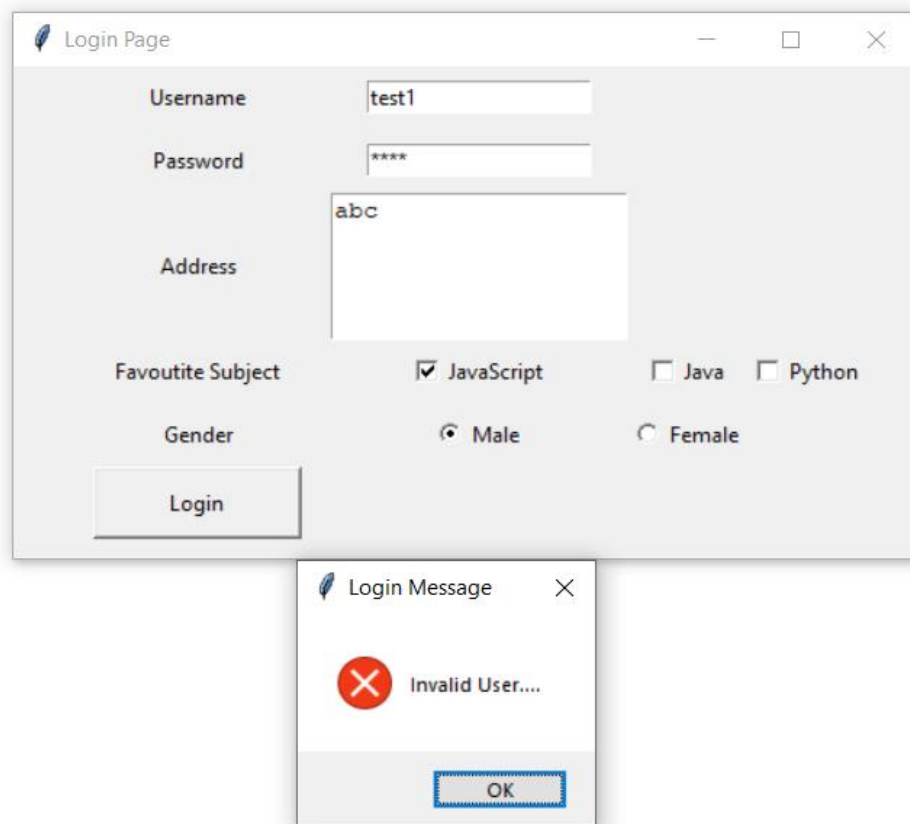
A screenshot of the "Login Page" window after input. The fields are filled with the following values:

- Username:** test
- Password:** \*\*\*\*
- Address:** abc
- Favourite Subject:** ☒ JavaScript, ☐ Java, ☐ Python
- Gender:** ☒ Male, ☐ Female

The "Login" button is still visible. Overlaid on top of the bottom right of the "Login Page" window is a smaller dialog box titled "Login Message". This dialog box contains:

- An information icon (a blue circle with a white 'i').
- The text: "Welcome, Login Successful..."
- An "OK" button at the bottom right.

### If Invalid Input:



### CONCLUSION:

In this experiment, I learnt about tkinter module, a standard Python interface to the Tk GUI toolkit shipped with Python. With the help of tkinter, I embedded different widgets like labels, text area, radio box, checkbox, and button on the Login page. The Login button also had a trigger associated with it and it checks for user validation, i.e., whether you are the user, if yes then a pops up 'Welcome. Login Successful..' otherwise 'Invalid User'. The widgets have different parameters through which we can set the different properties and styling of that widget.

## EXPERIMENT 9

**AIM:** Program to demonstrate CRUD (create, read, update and delete) operations on database (SQLite/ MySQL) using python

### THEORY:

#### Sqlite Database

SQLite is a self-contained transactional relational database engine that doesn't require a server configuration, as in the case of Oracle, MySQL, etc. The entire SQLite database is contained in a single file, which can be put anywhere in the computer's file system. SQLite is widely used as an embedded database in mobile devices, web browsers and other stand-alone applications. In spite of being small in size, it is a fully ACID compliant database conforming to ANSI SQL standards.

#### Python DB-API

Python Database API is a set of standards recommended by a Special Interest Group for database module standardization. Python modules that provide database interfacing functionality with all major database products are required to adhere to this standard. DB-API standards.

Standard Python distribution has in-built support for SQLite database connectivity. It contains sqlite3 module which adheres to DB-API 2.0. Other RDBMS compliant modules to DB-API:

- MySQL: PyMySQL module
- Oracle: Cx-Oracle module
- SQL Server: PyMsSql module
- PostgreSQL: psycopg2 module
- ODBC: pyodbc module

In order to establish a connection with a SQLite database, sqlite3 module needs to be imported and the **connect()** function needs to be executed.

```
>>> import sqlite3  
>>> db=sqlite3.connect('test.db')
```

The following methods are defined in the connection class:

Method	Description
cursor()	Returns a Cursor object which uses this Connection.
commit()	Explicitly commits any pending transactions to the database. The method should be a no-op if the underlying db does not support transactions.
rollback()	This optional method causes a transaction to be rolled back to the starting point. It may not be implemented everywhere.
close()	Closes the connection to the database permanently. Attempts to use the connection after calling this method will raise a DB-API Error.

A **cursor** is a Python object that enables one to work with the database. It acts as a handle for a given SQL query; it allows the retrieval of one or more rows of the result. A cursor object is obtained from the connection to execute SQL queries using the following statement:

```
>>>cur = db.cursor()
```

The following methods of the cursor object are:

Method	Description
execute()	Executes the SQL query in a string parameter
executemany()	Executes the SQL query using a set of parameters in the list of tuples
fetchone()	Fetches the next row from the query result set.
fetchall()	Fetches all remaining rows from the query result set.
callproc()	Calls a stored procedure.
close()	Closes the cursor object.

## CRUD Operations:

### 1) CREATE - a New Table

A string enclosing the CREATE TABLE query is passed as parameter to the execute() method of the cursor object.

### 2) Insert a New Record

The execute() method of the cursor object should be called with a string argument representing the INSERT query syntax. For example in a student table having three fields: name, age and marks. The string holding the INSERT query is defined as:

```
>>> qry="INSERT INTO student (name, age, marks) VALUES ('Rajeev',20,50);"
```

### 3) Insert multiple records

executemany() method is used to add multiple records at once. Data to be added should be given in a list of tuples, with each tuple containing one record. The list object (containing tuples) is the parameter of the executemany() method, along with the query string. For example,

```
>>> qry="insert into student (name, age, marks) values(?,?,?);"
>>> students=[('Jill', 18, 70), ('Jane', 25, 87)]
.....
>>> cur=db.cursor()
>>> cur.executemany(qry, students)
>>> db.commit()
```

### 4) READ - Retrieve Records

When the query string holds a SELECT query, the execute() method forms a result set object containing the records returned. Python DB-API defines two methods to fetch the records:

- fetchone(): Fetches the next available record from the result set. It is a tuple consisting of values of each column of the fetched record.

```
record = cur.fetchone()
```

- fetchall(): Fetches all remaining records in the form of a list of tuples. Each tuple corresponds to one record and contains values of each column in the table.

```
records = cur.fetchall()
```

### 5) Update a Record

The query string in the execute() method should contain an UPDATE query syntax. For example, to update the value of 'age' to 17 for 'John'

```
>>> qry="update student set age = 17 where name='John' ; "
```

## 6) Delete a Record

The query string should contain the DELETE query syntax. For example, the below code is used to delete 'Bill' from the student table.

```
>>> qry="DELETE from student where name='Bill';"
```

### CODE:

```
import sqlite3
```

```
conn = sqlite3.connect("test.db")  
crsr = conn.cursor()  
print("Connected to database")
```

```
#Creating table
```

```
cmd = "CREATE TABLE Employee(empId INTEGER PRIMARY KEY  
AUTOINCREMENT, fname VARCHAR(20) NOT NULL, lname VARCHAR(20)  
NOT NULL, gender CHAR(1) DEFAULT 'M');"  
crsr.execute(cmd)
```

```
#Inserting data
```

```
cmd = "INSERT INTO Employee(fname,lname,gender) VALUES ('Jack','Smith','M'),  
('Jane','Doyle', 'F'), ('Tim','Brookes','M'), ('Lydia','Simson','F'), ('Penny','Hill','M');"  
crsr.execute(cmd)  
conn.commit()
```

```
#Reading data
```

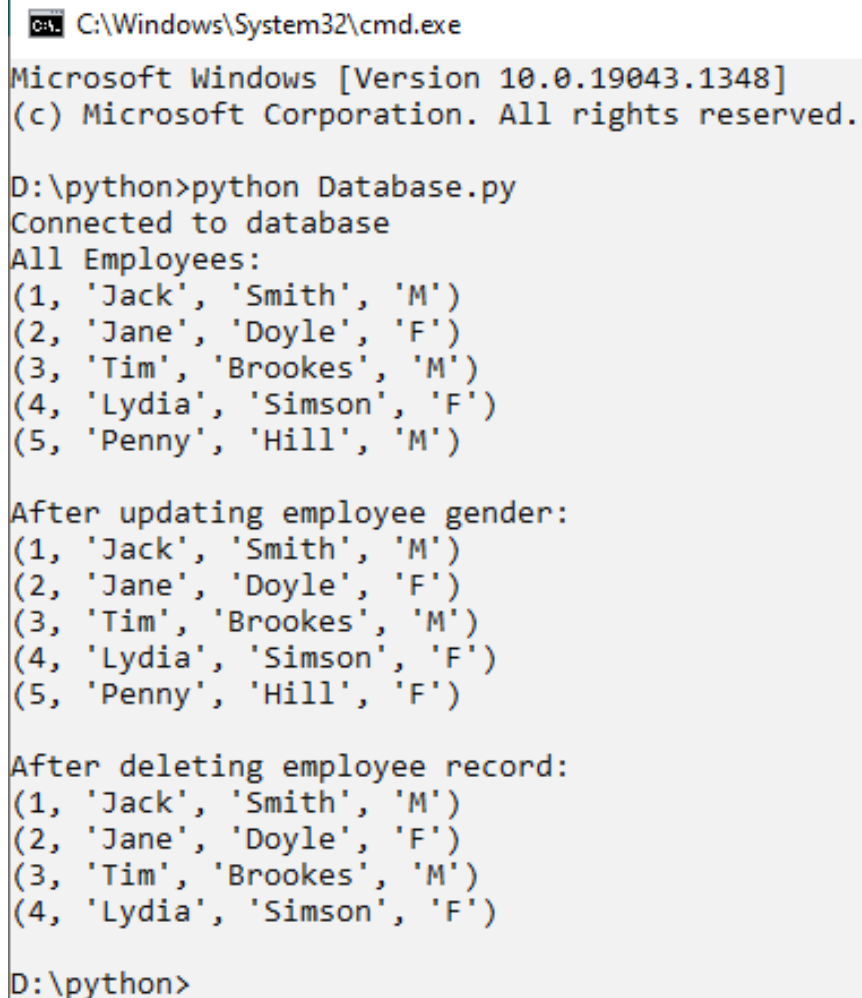
```
print("All Employees:")  
cmd = "SELECT * FROM Employee;"  
crsr.execute(cmd)  
results = crsr.fetchall()  
for emp in results :  
    print(emp)
```

```
#Updating data
```

```
crsr = conn.cursor()  
cmd = "UPDATE Employee SET gender='F' WHERE fname='Penny' AND  
lname='Hill';"  
crsr.execute(cmd)  
conn.commit()  
print("\nAfter updating employee gender: ")  
cmd = "SELECT * FROM Employee;"  
crsr.execute(cmd)  
results = crsr.fetchall()
```

```
for emp in results :  
    print(emp)  
  
#Deleting data  
crsr = conn.cursor()  
cmd = "DELETE FROM Employee WHERE fname='Penny';"  
crsr.execute(cmd)  
conn.commit()  
print("\nAfter deleting employee record:")  
cmd = "SELECT * FROM Employee;"  
crsr.execute(cmd)  
results = crsr.fetchall()  
for emp in results :  
    print(emp)  
  
conn.close()
```

## OUTPUT:



```
C:\Windows\System32\cmd.exe  
Microsoft Windows [Version 10.0.19043.1348]  
(c) Microsoft Corporation. All rights reserved.  
  
D:\python>python Database.py  
Connected to database  
All Employees:  
(1, 'Jack', 'Smith', 'M')  
(2, 'Jane', 'Doyle', 'F')  
(3, 'Tim', 'Brookes', 'M')  
(4, 'Lydia', 'Simson', 'F')  
(5, 'Penny', 'Hill', 'M')  
  
After updating employee gender:  
(1, 'Jack', 'Smith', 'M')  
(2, 'Jane', 'Doyle', 'F')  
(3, 'Tim', 'Brookes', 'M')  
(4, 'Lydia', 'Simson', 'F')  
(5, 'Penny', 'Hill', 'F')  
  
After deleting employee record:  
(1, 'Jack', 'Smith', 'M')  
(2, 'Jane', 'Doyle', 'F')  
(3, 'Tim', 'Brookes', 'M')  
(4, 'Lydia', 'Simson', 'F')  
  
D:\python>
```



## **CONCLUSION:**

In this experiment, I have implemented CRUD i.e. Create, Read, Update and Delete operations on SQLite database using python. Python DB-API provides database interfacing functionality with all major database products adhering to this standard. DB-API is also compatible with MySQL: PyMySQL module Cx-Oracle module, SQL Server: PyMsSql module, PostgreSQL and ODBC. SQLite Database is a self-contained transactional relational database engine that is used as an embedded database in python. Once the connection is established all the operations are controlled by the cursor. After writing a SQLite query we just need to pass it to execute method and it executes the following query. Thus, I have implemented the CRUD operations by creating a table, inserting one and many rows, reading, updating and deleting the employee table.

## EXPERIEMNT 11

**AIM:** Make use of advance modules of Python like OpenCV, Matplotlib, NumPy

### **THEORY:**

#### **OpenCV:**

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc. OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 18 million. The library is used extensively in companies, research groups and by governmental bodies.

Along with well-established companies like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota that employ the library, there are many startups such as Applied Minds, VideoSurf, and Zeitera, that make extensive use of OpenCV. OpenCV's deployed uses span the range from stitching streetview images together, detecting intrusions in surveillance video in Israel, monitoring mine equipment in China, helping robots navigate and pick up objects at Willow Garage, detection of swimming pool drowning accidents in Europe, running interactive art in Spain and New York, checking runways for debris in Turkey, inspecting labels on products in factories around the world on to rapid face detection in Japan.

It has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. OpenCV leans mostly towards real-time vision applications and takes advantage of MMX and SSE instructions when available. A full-featured CUDA and OpenCL interfaces are being actively developed right now. There are over 500 algorithms and about 10 times as many functions that compose or support those algorithms. OpenCV is written natively in C++ and has a templated interface that works seamlessly with STL containers.

## Matplotlib:

Matplotlib is an amazing visualization library in Python for 2D plots of arrays. Matplotlib is a multi-platform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack. It was introduced by John Hunter in the year 2002.

One of the greatest benefits of visualization is that it allows us visual access to huge amounts of data in easily digestible visuals. Matplotlib consists of several plots like line, bar, scatter, histogram etc.

## NumPy:

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the *ndarray* object. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- 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. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

## CODE:

### 1. OpenCV:

#### A) Read Image

```
▶ from google.colab.patches import cv2_imshow  
import cv2  
image = cv2.imread("img.jpg")  
print(image)
```

```
[[[172 112 60]  
 [172 112 60]  
 [172 112 60]  
 ...  
 [186 203 224]  
 [187 204 225]  
 [187 204 225]]]
```

```
[[[172 112 60]  
 [172 112 60]  
 [172 112 60]  
 ...  
 [188 205 226]  
 [189 206 227]  
 [189 206 227]]]
```

```
[[[172 112 60]  
 [172 112 60]  
 [172 112 60]  
 ...  
 [191 205 227]  
 [192 206 228]  
 [193 207 229]]]
```

```
...  
[[ [23 73 71]  
 [ 20 75 72]  
 [ 25 91 86]  
 ...  
 [ 8 98 98]  
 [ 14 104 104]  
 [ 10 103 102]]]
```

```
[[ 12 62 58]
 [ 6 60 55]
 [ 2 66 60]
 ...
 [ 10 96 96]
 [ 17 103 103]
 [ 11 99 99]]

[[ 0 47 44]
 [ 0 52 47]
 [ 5 67 61]
 ...
 [ 11 94 95]
 [ 21 104 105]
 [ 16 102 102]]]
```

## B) Display Image



### C) Draw Line, Rectangle, Circle on the image

```
[ ] import cv2
    from google.colab.patches import cv2_imshow
    img = cv2.imread('logo.png')
    img = cv2.line(img , (20,20) , (380,500) , (100,0,50) , 8)
    mg = cv2.rectangle(img , (320,100) , (400,300) , (50,150,50) , 8)
    mg = cv2.circle(img , (100,100) , 35 , (0,10,250) , 8)
    cv2_imshow(img)
```



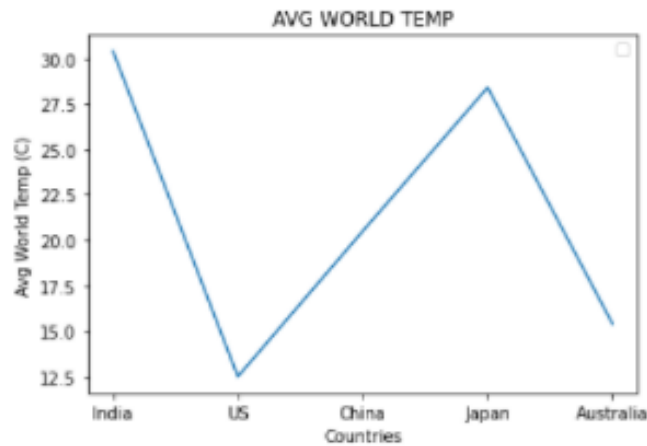
## 2. Matplotlib:

### A) Line Plot

```
[ ] from matplotlib import pyplot as plt
    import numpy as np

    countries = ["India" , "US" , "China" , "Japan" , "Australia"]
    temp = [30.4, 12.5, 20.5, 28.4, 15.4]

    plt.plot(countries, temp)
    plt.xlabel('Countries')
    plt.ylabel("Avg World Temp (C)")
    plt.title("AVG WORLD TEMP")
    plt.legend()
    plt.show()
```



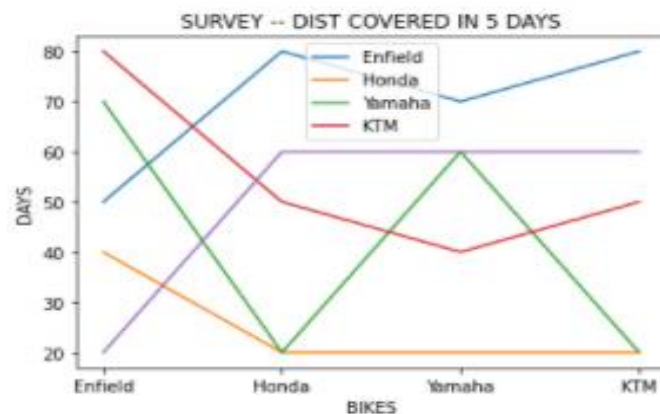
## B) Multi-Line Plot

```
[ ] from matplotlib import pyplot as plt
import numpy as np

days = ["day1", "day2", "day3", "day4", "day5"]
bike = ["Enfield", "Honda", "Yamaha", "KTM"]
d1 = [50, 80, 70, 80]
d2 = [40, 20, 20, 20]
d3 = [70, 20, 60, 20]
d4 = [80, 50, 40, 50]
d5 = [20, 60, 60, 60]

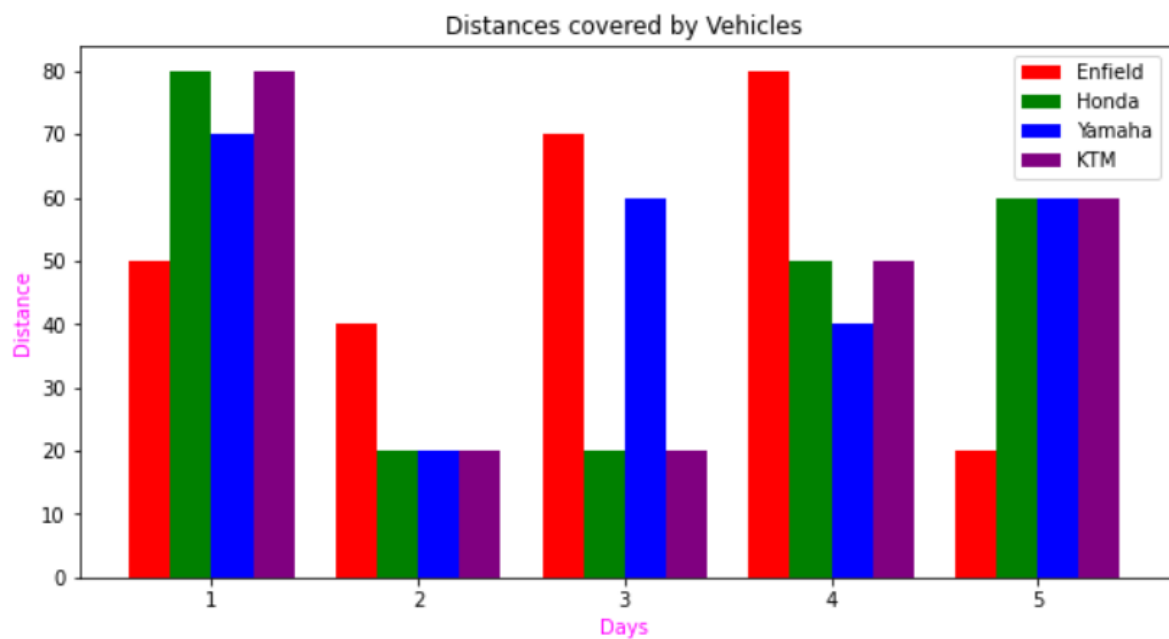
plt.plot(bike, d1, label="DAY1")
plt.plot(bike, d2, label="DAY2")
plt.plot(bike, d3, label="DAY3")
plt.plot(bike, d4, label="DAY4")
plt.plot(bike, d5, label="DAY5")

plt.xlabel('BIKES')
plt.ylabel("DAYS")
plt.title("SURVEY -- DIST COVERED IN 5 DAYS")
plt.legend(bike)
plt.show()
```



### 3. Bar Plot

```
▶ parent = [enfield, honda, yamaha, ktm]
names = ["Enfield", "Honda", "Yamaha", "KTM"]
colors = ["Red", "Green", "Blue", "Purple"]
u = 0
plt.figure(figsize=(10, 5))
for values, name, color in zip(parent, names, colors):
    plt.bar(days + u, values, width=0.2, color=color)
    plt.title("Distances covered by Vehicles")
    plt.xlabel("Days", color="magenta")
    plt.ylabel("Distance", color="magenta")
    u+= 0.2
plt.xticks(days + 0.6/2, days)
plt.legend(names)
plt.show()
```





## 4. NumPy:

```
[7] from numpy import *  
  
array = array([1, 2, 3, 4, 5, 6], int)  
print(array)  
  
[1 2 3 4 5 6]
```

```
[8] print(array[::-1])  
print(array[-1:-5:-1])  
  
[6 5 4 3 2 1]  
[6 5 4 3]
```

```
[9] b = array  
b[1] = 700  
print(b, array)  
  
[ 1 700  3  4  5  6] [ 1 700  3  4  5  6]
```

```
[11] c = array.view()  
c[3] = 600  
print(c, b, array)  
  
[ 1 700  3 600  5  6] [ 1 700  3 600  5  6] [ 1 700  3 600  5  6]
```

```
[12] print(id(b), id(c), id(array))  
  
140402161322304 140402161662672 140402161322304
```

```
import numpy as np  
first_matrix = np.arange(3, 12).reshape(3,3)  
print(first_matrix)
```

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

```
] second_matrix = np.square(first_matrix)  
print(second_matrix)
```

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

```
[ ] print("MAX: ",np.max(first_matrix))  
    print("MIN: ",np.min(first_matrix))  
    print("MEAN: ",np.mean(first_matrix))  
    print("SUM: ",np.sum(first_matrix))
```

```
MAX:  11  
MIN:  3  
MEAN:  7.0  
SUM:  63
```

## CONCLUSION:

In this experiment, I learnt about advance modules of Python like OpenCV, Matplotlib, NumPy. Using OpenCV we can load, read images into python program. We can also perform transformation on the image and save it. Also using OpenCV we can make shapes on the images. Using Matplotlib we can plot various graphs like line chart, bar plot used for data visualization. We can also set labels, ticks, title, grid, scales of the chart. Using Numpy we can make arrays and these arrays have many built-in methods along with them. Numpy arrays can be reshaped to multi-dimensional arrays and even flatten to 1D array. Thus, in this experiment I explored methods of these advance python modules and implemented the codes for the same.

## EXPERIEMNT 12

**AIM:** Creating web application using Django web framework to demonstrate functionality of user login and registration (also validating user detail using regular expression).

### **THEORY:**

Django is a high-level Python web framework that enables rapid development of secure and maintainable websites. Built by experienced developers, Django takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It is free and open source, has a thriving and active community, great documentation, and many options for free and paid-for support.

Django helps you write software that is:

#### *Complete*

Django follows the "Batteries included" philosophy and provides almost everything developers might want to do "out of the box". Because everything you need is part of the one "product", it all works seamlessly together, follows consistent design principles, and has extensive and up-to-date documentation

#### *Versatile*

Django can be (and has been) used to build almost any type of website — from content management systems and wikis, through to social networks and news sites. It can work with any client-side framework, and can deliver content in almost any format (including HTML, RSS feeds, JSON, XML, etc). The site you are currently reading is built with Django!

Internally, while it provides choices for almost any functionality you might want (e.g. several popular databases, templating engines, etc.), it can also be extended to use other components if needed.

#### *Secure*

Django helps developers avoid many common security mistakes by providing a framework that has been engineered to "do the right things" to protect the website automatically. For example, Django provides a secure way to manage user accounts and passwords, avoiding common mistakes like putting session information in cookies where it is vulnerable (instead cookies just contain a key, and the actual data is stored in the database) or directly storing passwords rather than a password hash.

A password hash is a fixed-length value created by sending the password through a cryptographic hash function. Django can check if an entered password is correct by

running it through the hash function and comparing the output to the stored hash value. However due to the "one-way" nature of the function, even if a stored hash value is compromised it is hard for an attacker to work out the original password.

Django enables protection against many vulnerabilities by default, including SQL injection, cross-site scripting, cross-site request forgery and clickjacking (see Website security for more details of such attacks)

### *Scalable*

Django uses a component-based “shared-nothing” architecture (each part of the architecture is independent of the others, and can hence be replaced or changed if needed). Having a clear separation between the different parts means that it can scale for increased traffic by adding hardware at any level: caching servers, database servers, or application servers. Some of the busiest sites have successfully scaled Django to meet their demands (e.g. Instagram and Disqus, to name just two).

### *Maintainable*

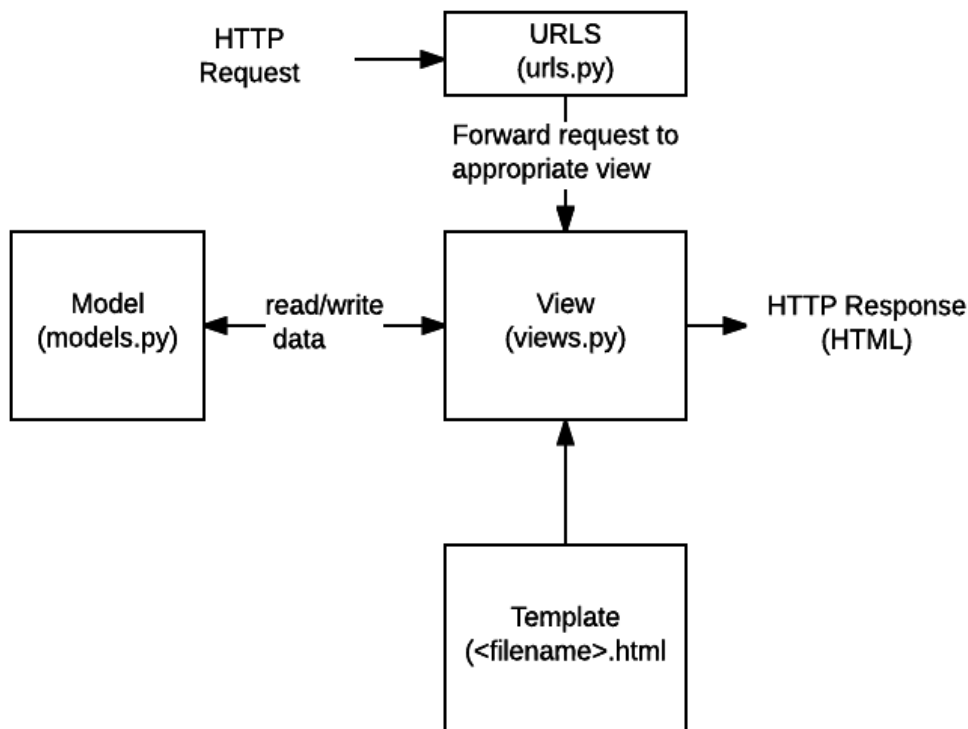
Django code is written using design principles and patterns that encourage the creation of maintainable and reusable code. In particular, it makes use of the Don't Repeat Yourself (DRY) principle so there is no unnecessary duplication, reducing the amount of code. Django also promotes the grouping of related functionality into reusable "applications" and, at a lower level, groups related code into modules (along the lines of the Model View Controller (MVC) pattern).

### *Portable*

Django is written in Python, which runs on many platforms. That means that you are not tied to any particular server platform, and can run your applications on many flavours of Linux, Windows, and Mac OS X. Furthermore, Django is well-supported by many web hosting providers, who often provide specific infrastructure and documentation for hosting Django sites.

In a traditional data-driven website, a web application waits for HTTP requests from the web browser (or other client). When a request is received the application works out what is needed based on the URL and possibly information in POST data or GET data. Depending on what is required it may then read or write information from a database or perform other tasks required to satisfy the request. The application will then return a response to the web browser, often dynamically creating an HTML page for the browser to display by inserting the retrieved data into placeholders in an HTML template.

Django web applications typically group the code that handles each of these steps into separate files:



Django - files for views, model, urls, template

**URLs:** While it is possible to process requests from every single URL via a single function, it is much more maintainable to write a separate view function to handle each resource. A URL mapper is used to redirect HTTP requests to the appropriate view based on the request URL. The URL mapper can also match particular patterns of strings or digits that appear in a URL and pass these to a view function as data.

**View:** A view is a request handler function, which receives HTTP requests and returns HTTP responses. Views access the data needed to satisfy requests via models, and delegate the formatting of the response to templates.

**Models:** Models are Python objects that define the structure of an application's data, and provide mechanisms to manage (add, modify, delete) and query records in the database.

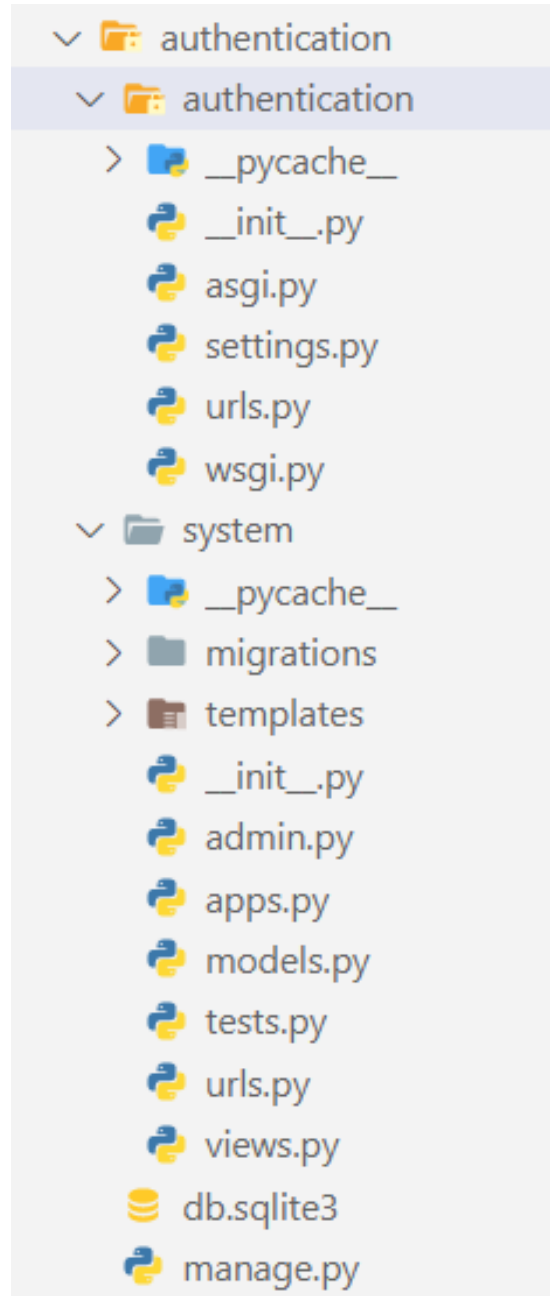
**Templates:** A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. A view can dynamically create an HTML page using an HTML template, populating it with data from a model. A template can be used to define the structure of any type of file; it doesn't have to be HTML.

## CODE:

**Project Name :** authentication

**App Name :** system

**Folder Structure :**



### **authentication/system/views.py:**

```
from django.contrib import auth
from django.shortcuts import redirect, render
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.models import User
```

```
from django.contrib import messages
import re

# Create your views here.

def homeRenderer(request):
    if request.method == "GET":
        return render(request, "system/login.html")

    elif request.method == "POST":
        username = request.POST.get("username")
        password = request.POST.get("password")

        user = authenticate(username=username, password=password)

        if user is None:
            messages.add_message(request, messages.ERROR, "Login Failed! User not
found.")
            return render(request, "system/login.html")

        login(request, user)

        return render(request, "system/login.html")

def registerRenderer(request):
    if request.method == "GET":
        return render(request, "system/register.html")

    elif request.method == "POST":
        username = request.POST.get("username")
        password = request.POST.get("password")
        confirm = request.POST.get("confirm")
        if re.search(r'^[A-Za-z][A-Za-z0-9_]', username):# Regex check implemented
here
            if password!=confirm:
                messages.add_message(request, messages.ERROR, "Passwords do not
match")
                return render(request, "system/register.html")

            if User.objects.filter(username=username).exists():
                messages.add_message(request, messages.ERROR, "Username Exists")
                return render(request, "system/register.html")

            User.objects.create_user(username=username, password=password)
```

```
user = authenticate(username=username, password=password)

if user is None:
    messages.add_message(request, messages.ERROR, "Login Failed")
    return render(request, "system/login.html")

login(request, user)

return redirect("/")
else:
    messages.add_message(request, messages.ERROR, "Invalid Username
Format")
    return render(request, "system/register.html")

def sessionEnd(request):
    if request.user.is_authenticated:
        logout(request)
    return redirect("/")
```

### **authentication/system/urls.py**

```
from . import views
from django.urls import path

urlpatterns = [
    path("", views.homeRenderer, name="home"),
    path("register/", views.registerRenderer, name="register"),
    path("logout/", views.sessionEnd, name="logout")
]
```

### **authentication/authentication/settings.py**

```
"""
```

Django settings for authentication project.

Generated by 'django-admin startproject' using Django 3.2.8.

For more information on this file, see  
<https://docs.djangoproject.com/en/3.2/topics/settings/>

For the full list of settings and their values, see  
<https://docs.djangoproject.com/en/3.2/ref/settings/>  
"""

```
from pathlib import Path
```



```
# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent

# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/3.2/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'django-insecure-s-78msca19^20e2&-nyb6q-
r@5@ka^7&q3z^ke7+jkij5f&u1#'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []

# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'system',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

ROOT_URLCONF = 'authentication.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
```

```
'OPTIONS': {
    'context_processors': [
        'django.template.context_processors.debug',
        'django.template.context_processors.request',
        'django.contrib.auth.context_processors.auth',
        'django.contrib.messages.context_processors.messages',
    ],
},
],
```

```
WSGI_APPLICATION = 'authentication.wsgi.application'
```

```
# Database
```

```
# https://docs.djangoproject.com/en/3.2/ref/settings/#databases
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

```
# Password validation
```

```
# https://docs.djangoproject.com/en/3.2/ref/settings/#auth-password-validators
```

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME':
'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]
```

```
# Internationalization
```

```
# https://docs.djangoproject.com/en/3.2/topics/i18n/
```

```
LANGUAGE_CODE = 'en-us'
```

```
TIME_ZONE = 'UTC'
```

```
USE_I18N = True
```

```
USE_L10N = True
```

```
USE_TZ = True
```

```
# Static files (CSS, JavaScript, Images)
```

```
# https://docs.djangoproject.com/en/3.2/howto/static-files/
```

```
STATIC_URL = '/static/'
```

```
# Default primary key field type
```

```
# https://docs.djangoproject.com/en/3.2/ref/settings/#default-auto-field
```

```
DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
```

## **authentication/authentication/urls.py**

```
"""authentication URL Configuration
```

The `urlpatterns` list routes URLs to views. For more information please see:

<https://docs.djangoproject.com/en/3.2/topics/http/urls/>

Examples:

Function views

1. Add an import: `from my_app import views`
2. Add a URL to `urlpatterns`: `path("", views.home, name='home')`

Class-based views

1. Add an import: `from other_app.views import Home`
2. Add a URL to `urlpatterns`: `path("", Home.as_view(), name='home')`

Including another `URLconf`

1. Import the `include()` function: `from django.urls import include, path`
2. Add a URL to `urlpatterns`: `path('blog/', include('blog.urls'))`

```
"""
```

```
from django.contrib import admin
```

```
from django.urls import path, include
```

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path("", include("system.urls")),  
]
```

## OUTPUT:

1. Regex Check for invalid username (username cannot start with a numeric value):

<input type="text" value="1234"/>	<input type="text" value="Username"/>
<input type="text" value="..."/>	<input type="text" value="Password"/>
<input type="text" value="..."/>	<input type="text" value="Confirm Password"/>
<input type="button" value="Register"/>	<input type="button" value="Register"/>

Invalid Username Format

2. Valid Registration:

<input type="text" value="debian"/>	<b>Logged In as debian</b>
<input type="text" value="..."/>	
<input type="text" value="..."/>	
<input type="button" value="Register"/>	
	<input type="button" value="Logout"/>

3. Valid User Login

<input type="text" value="debian"/>	<b>Logged In as debian</b>
<input type="text" value="..."/>	
<input type="button" value="Login"/>	
<input type="button" value="Register"/>	
	<input type="button" value="Logout"/>

## CONCLUSION:

In this experiment I have successfully implemented the login and registration (authentication) mechanism using the Django Framework. The credentials of the user were also validated using RE (Regular Expression) module which has been previously studied in the course.