UNIT-4

Syllabus:

File Operations: Understanding read functions, read (), readline () and readlines (), Understanding write functions, write () and writelines (), Manipulating file pointer using seek, Programming using file operations, Reading config files in python, Writing log files in python.

Object Oriented Programming: Concept of class, object and instances, Constructor, class attributes and destructors, Real time use of class in live projects, Inheritance, overlapping and overloading operators, Adding and retrieving dynamic attributes of classes, Programming using Oops support.

Design with Classes: Objects and Classes, Data modelling Examples, Case Study An ATM, Structuring Classes with Inheritance and Polymorphism.

Files in Python:

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files. Python provides us with an important feature for reading data from the file and writing data into a file. Mostly, in programming languages, all the values or data are stored in some variables which are volatile in nature. Because data will be stored into those variables during run-time only and will be lost once the program execution is completed. Hence it is better to save these data permanently using files. Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object.

Opening and Closing Files

The open () Method

Before you can read or write a file, you have to open it using Python's built-in open () function. This function creates a file object, which would be utilized to call other support methods associated with it.

Syntax: file object = open (filename, access mode)

Here are parameter details –

file_name − The file_name argument is a string value that contains the name of the file that you want to access.

access_mode − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

Here is a list of the different modes of opening a file –

Sno	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb** Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

6	**wb** Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	**w+** Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	**wb+** Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	**a** Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	**ab** Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	**a+** Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	**ab+** Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The file Object Attributes:

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object −

Sno	**Attribute & Description**
1	**file.closed** Returns true if file is closed, false otherwise.
2	**file.mode** Returns access mode with which file was opened.
3	**file.name** Returns name of the file.

Example:

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file

#File object attributes

print('Name of the file: ', f.name)

print('Closed or not : ', f.closed)

print('Opening mode : ', f.mode)

f.close()
```

The close () Method

The close () method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done. It is a good practice to use the close () method to close a file.

Syntax: fileObject.close()

Example:

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file

#File object attributes

print('Name of the file: ', f.name)

print('Closed or not : ', f.closed)

print('Opening mode : ', f.mode)

f.close()
```

Reading and Writing Files

The file object provides a set of access methods. Now, we will see how to use read (), readline (), readlines () and write (), writelines () methods to read and write files.

Understanding write () and writelines ()

The write () Method

- The write () method writes any string (binary data and text data) to an open file.
- The write () method does not add a newline character ('\n') to the end of the string

Syntax: fileObject.write(string)

Here, passed parameter is the content to be written into the opened file.

Example:

```
f=open('sample.txt','w') #creates a new file sample.txt give write permissions on file

#writing content into file sample.txt using write method

f.write( "Python is a great language.")

f.close()
```

The writelines () method:

Python file method **writelines ()** writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value .

Syntax: fileObject.writelines(sequence)

Parameters

Sequence – This is the Sequence of the strings.

Return Value-This method does not return any value.

Example

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file

#writing content into file using write method

f.writelines (['python is easy\n','python is portable\n','python is  comfortable']

)

 f.close()
```

Understanding read (), readline () and readlines ():

The read () Method

The read () method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax: fileObject.read([count])

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file
#writing content into file using write method

f.writelines(['python is easy\n','python is portable\n','python is comfortable'])

 f.close()

f=open('sample.txt','r')

#reading first 20 bytes from the file using read() method

print(f.read(20))
```

The readline () Method

Python file method **readline()**reads one entire line from the file. A trailing newline character is kept in the string. If the *size* argument is present and non-negative, it is a maximum byte count including the trailing newline and an incomplete line may be returned.

An empty string is returned only when EOF is encountered immediately.

Syntax: fileObject.readline(size)

Parameters

- **size** − This is the number of bytes to be read from the file.

Return Value

- This method returns the line read from the file.

Example

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file

#writing content into file using write method

f.writelines(['python is easy\n','python is portable\n','python is comfortable'])

f.close()
```

```
f=open('sample.txt','r')

#reading first line of the file using readline() method
print(f.readline())
```

The readlines () Method

Python file method **readlines()** reads until EOF using readline() and returns a list containing the lines. If the optional *sizehint* argument is present, instead of reading up to EOF, whole lines totalling approximately *sizehint* bytes (possibly after rounding up to an internal buffer size) are read.

An empty string is returned only when EOF is encountered immediately.

Syntax: fileObject.readlines(sizehint)

Parameters

- **sizehint** − This is the number of bytes to be read from the file.

Return Value

This method returns a list containing the lines.

Example

```
f=open('sample.txt','w') # creates a new file sample.txt give write permissions on file

#writing content into file using write method

f.writelines(['python is easy\n','python is portable\n','python is comfortable'])

 f.close()

f=open('sample.txt','r')

#reading all the line of the file using readlines() method

print(f.readlines())
```

Manipulating file pointer using seek():

tell (): The tell () method tells you the current position within the file

Syntax: file_object.tell()

Example:

```
# Open a file

fo = open("sample.txt", "r+")

str = fo.read(10)

print("Read String is : ", str)

# Check current position

position = fo.tell()

print("Current file position : ", position)

fo.close()
```

seek (): The seek (offset, from_what) method changes the current file position.

Syntax: f.seek(offset, from_what) #where f is file pointer

Parameters:

Offset: Number of postions to move forward

from_what: It defines point of reference.

Returns: Does not return any value

The reference point is selected by the **from_what** argument. It accepts three values:

0: sets the reference point at the beginning of the file

1: sets the reference point at the current file position

2: sets the reference point at the end of the file

By default from_what argument is set to 0.

Note: Reference point at current position / end of file cannot be set in text mode except when offset is equal to 0.

Example:

```
# Open a file
```

```
fo = open("sample.txt", "r+")

str = fo.read(10)
print("Read String is : ", str)

# Check current position

position = fo.tell()

print("Current file position : ", position)

# Reposition pointer at the beginning once again

position = fo.seek(0, 0);

str = fo.read(10)

print("Again read String is : ", str)

# Close opend file

fo.close()
```

File processing operations:

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

i) os.rename(): The rename() method takes two arguments, the current filename and the new filename.(to rename file)

Syntax: os.rename(current_file_name, new_file_name)

Example:

```
import os

os.rename('sample.txt','same.txt')
```

ii) os.mkdir(): The mkdir() method takes one argument as directory name, that you want to create.(This method is used to create directory)

Syntax: os.mkdir(directory name)
Example:

```
import os

os.mkdir('python') # Creates python named directory
```

iii) os.rmdir(): The rmdir() method takes one argument as directory name, that you want to remove.(This method is used to remove directory)

Syntax: os.rmdir(directory name)

Example:

```
import os

os.rmdir('python') # removes python named directory
```

iv) os.chdir(): The chdir() method takes one argument as directory name which we want to change.(This method is used to change directory)

Syntax: os.chdir(newdir)

Example:

```
import os

os.chdir('D:\>') # change directory to D drive
```

os.remove(): The remove() method takes one argument, the filename that you want to remove.(This method is used to remove file)

Syntax: os.remove(filename)

Example:

```
import os

os.remove('python,txt') # removes python.txt named file
```

os.getcwd(): The getcwd() method takes zero arguments,it gives current working director.

Syntax: os.getcwd()

Example:

```
import os
os.getcwd( ) # it gives current working directory
```

WRITING AND READING CONFIG FILES IN PYTHON

Config files help creating the initial settings for any project, they help avoiding the hardcoded data. For example, imagine if you migrate your server to a new host and suddenly your application stops working, now you have to go through your code and search/replace IP address of host at all the places. Config file comes to the rescue in such situation. You define the IP address key in config file and use it throughout your code. Later when you want to change any attribute, just change it in the config file. So this is the use of config file.

Creating and writing config file in Python

In Python we have configparser module which can help us with creation of config files (.ini format).

Program:

```
from configparser import ConfigParser

#Get the configparser object

config_object = ConfigParser()

#Assume we need 2 sections in the config file, let's call them USERINFO and SERVERCONFIG

config_object["USERINFO"] = {

 "admin": "Chankey Pathak",

 "loginid": "chankeypathak",

 "password": "tutswiki"

}

config_object["SERVERCONFIG"] = {
 "host": "tutswiki.com",

 "port": "8080",

 "ipaddr": "8.8.8.8"
}

#Write the above sections to config.ini file

with open('config.ini', 'w') as conf:

 config_object.write(conf)
```

Now if you check the working directory, you will notice config.ini file has been created, below

is its content.

```
[USERINFO]

admin = Chankey Pathak

password = tutswiki

loginid = chankeypathak


[SERVERCONFIG]

host = tutswiki.com

ipaddr = 8.8.8.8

port = 8080
```

Reading a key from config file:

So we have created a config file, now in your code you have to read the configuration data so that you can use it by “keyname” to avoid hardcoded data, let’s see how to do that

Program:

```
from configparser import ConfigParser
```

```
#Read config.ini file

config_object = ConfigParser()

config_object.read("config.ini")

#Get the password
userinfo          =          config_object["USERINFO"]

print("Password  is{}".format(userinfo["password"]))
```

output:

```
Password is tutswiki
```

UNIT-4
PART-2

Object Oriented Programming: Concept of class, object and instances, Constructor, class attributes and destructors, Real time use of class in live projects, Inheritance, overlapping and overloading operators, Adding and retrieving dynamic attributes of classes, Programming using Oops support

Design with Classes: Objects and Classes, Data modelling Examples, Case Study An ATM, Structuring Classes with Inheritance and Polymorphism

Introduction

We have two programming techniques namely

1. Procedural-oriented programming technique
2. Object-oriented programming technique

Till now we have using the Procedural-oriented programming technique, in which our program is written using functions and block of statements which manipulate data. However a better style of programming is Object-oriented programming technique in which data and functions are combined to form a class. Object Oriented programming (OOP) is a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects. There are many object-oriented programming languages including JavaScript, C++, Java, and Python.

Classes and objects are the main aspects of object oriented programming.

Overview of OOP Terminology

- **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** − A variable that is defined inside a method and belongs only to the

current instance of a class.

- **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** − The creation of an instance of a class.
- **Method** − A special kind of function that is defined in a class definition.
- **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** − The assignment of more than one function to a particular operator.

Benefits of OOP

- OOP models complex things as reproducible, simple structures
- Reusable, OOP objects can be used across programs
- Allows for class-specific behavior through polymorphism
- Easier to debug, classes often contain all applicable information to them
- Secure, protects information through encapsulation

Classes:

1. Class is a basic building block in python
2. Class is a blue print or template of a object
3. A class creates a new data type
4. And object is instance(variable) of the class
5. In python everything is an object or instance of some class

   Example :

   All integer variables that we define in our program are instances of class int.

   ```
   >>> a=10
   >>> type(a)
   <class 'int'>
   ```
6. The python standard library based on the concept of classes and objects

<u>Defining a class:</u>

Python has a very simple syntax of defining a class.

<u>Syntax :</u>

```
Class class-name:
    Statement1
    Statement2
    Statement3
    -
    -
    -
    Statement
```

From the syntax, Class definition starts with the keyword class followed by class-name and a colon(:). The statements inside a class are any of these following

1. Sequential instructions
2. Variable definitions
3. Decision control statements
4. Loop statements
5. Function definitions

<u>Note :</u> the class members are accessed through class object

<u>Note :</u> class methods have access to all data contained in the instance of the object

<u>Creating objects:</u> (creating an object of a class is known as class instantiation)

• Once a class is defined, the next job is to create a object of that class. • The object can then access class variables and class methods using dot operator

<u>Syntax of object creation:</u>

Object-name=class-name()

• Syntax for accessing class members through the class object is
Object-name.class-member-name

Example :

```
class ABC:
    a=10
obj=ABC()
print(obj.a)
```

<u>self variable and class methods</u>:

- Self refers to the object itself (Self is a pointer to the class instance)
- Whenever we define a member function in a class always use a self as a first argument and give rest of the arguments
- Even if it doesn't take any parameter or argument you must pass self to a member function
- We do not give a value for this parameter, when call the method, python will provide it.
- The self in python is equivalent to the this pointer in c++

Example 1 :

```
class Person:

        pc=0            # Class varibles

        def  setFullName(self,fName,lName):

                self.fName=fName # instance variables

                self.lName=lName     # instance variables


        def  printFullName(self):

                print(self.fName," ",self.lName)

                print("Person number : ",self.pc) #access Classvariable

PName=Person()                    #Object PName created

PName.setFullName("vamsi","kurama")

PName.pc=7                        #Attribute pc of PName modified

PName.printFullName()

P=Person()                        #Object P created

P.setFullName("Surya","Vinti")

P.pc=23                           #Attribute pc of P modified

P.printFullName()
```

Output:

```
>>>

vamsi   kurama

Person number :  7

Surya   Vinti

Person number :  23
```

<u>Constructor method:</u>

A constructor is a special type of method (function) that is called when it instantiates an object of a class. The constructors are normally used to initialize (assign values) to the instance variables.

Creating a constructor: (The name of the constructor is always the _ _init_ _().)

The constructor is always written as a function called __init__(). It must always take as its first argument a reference to the instance being constructed.

While creating an object, a constructor can accept arguments if necessary. When you create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.

Every class must have a constructor, even if it simply relies on the default

constructor. Example:

```
class Person:
        pc=0            # Class varibles
        def __init__(self):
                print("Constructor initialised ")
                self.fName="XXXX"
                self.lName="YYYY"
        def setFullName(self,fName,lName):
                self.fName=fName # instance variables
                self.lName=lName    # instance variables

        def printFullName(self):
                print(self.fName," ",self.lName)
                print("Person number : ",self.pc) #access Classvariable

PName=Person()
PName.printFullName()
PName.setFullName("vamsi","kurama")
PName.pc=7
print("After setting Name:")
PName.printFullName()
```

Output:

```
>>>
Constructor initialised
XXXX   YYYY
Person number :  0
After setting Name:
vamsi   kurama
Person number :  7
```

Destructor:

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically. The _ _ del _ _ ( ) method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax of destructor declaration:

```
def __del__(self):
 # body of destructor
```

Note: A reference to objects is also deleted when the object goes out of reference or when the program ends.

Example 1: Here is the simple example of destructor. By using del keyword we deleted the all references of object 'obj', therefore destructor invoked automatically.

```
# Python program to illustrate destructor
class Employee:

    # Initializing
    def  __init__(self):
        print('Employee created.')

    # Deleting (Calling destructor)
    def  __del__(self):
        print('Destructor called, Employee deleted.')

obj = Employee()
del obj
```

Output:

```
Employee created
```

Destructor called, Employee deleted

<u>Inheritance:</u>

One of the major advantages of Object Oriented Programming is reusability. Inheritance is one of the mechanisms to achieve the reusability. Inheritance is used to implement is-a relationship.

Definition: A technique of creating a new class from an existing class is called inheritance. The old or existing class is called base class or super class and a new class is called sub class or derived class or child class.

The derived class inherits all the variable and methods of the base class and adds their own variables and methods. In this process of inheritance base class remains unchanged.

Syntax to inherit a class:

```
Class MySubClass(object):

        Pass(Body-of-the-derived-class)
```

Example :

```
class Pet:
        def __init__(self,name,age):
                self.name=name
                self.age=age
class Dog(Pet):
        def sound(self):
                print("I am {} and My age is {} and I sounds
                Like".format(self.name,self.age)) print("Bow Bow..")
class Cat(Pet):
        def sound(self):
                print("I am {} and My age is {} and I sounds
                Like".format(self.name,self.age)) print("Meow Meow..")
class Parrot(Pet):
        def sound(self):
                print("Hello I am {} and My age is {} ".format(self.name,self.age))
p1=Dog("Dozer",4)
p2=Cat("Edward",3)
p3=Parrot("Jango",6)
p1.sound()
p2.sound()
p3.sound()
```

Example 2:

```
class Person:
        def __init__(self,name,age):
                self.name=name
                self.age=age
        def display(self):
                print("name=",self.name)
                print("age=",self.age)
class Teacher(Person):
        def __init__(self,name,age,exp,r_area):
                Person.__init__(self,name,age)
                self.exp=exp
                self.r_area=r_area
        def displayData(self):
                Person.display(self)
                print("Experience=",self.exp)
                print("Research area=",self.r_area)
class Student(Person):
        def __init__(self,name,age,course,marks):
                Person.__init__(self,name,age)
                self.course=course
                self.marks=marks
        def displayData(self):
                Person.display(self)
                print("course=",self.course)
                print("marks=",self.marks)
print("********TEACHER***********")
t=Teacher("jai",55,13,"cloud computing")
t.displayData()
print("********STUDENT***********")
s=Student("hari",21,"B.Tech",99)
s.displayData()
```

Types of inheritance:

Python supports the following types of inheritan:

i) Single inheritance
ii) Multiple Inheritance
iii) Multi-level Inheritance
iv) Multi path Inheritance

Single Inheritance:

When a derived class inherits features form only one base class, it is called Single inheritance.

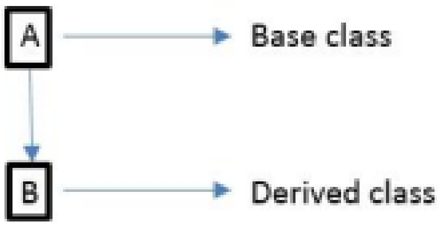
Syntax:

```
class Baseclass:

        <body of base class>

class Derivedclass(Baseclass):

        <body of the derived class>
```



Example:

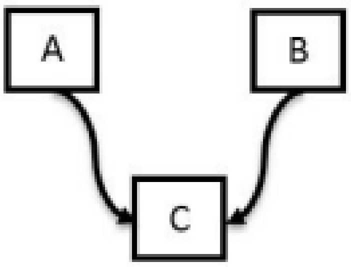
```
class A:
        i=10
class B(A):
        j=20
```

```
obj=B()
print("member of class A is",obj.i)
print("member of class B is",obj.j)
```

Multiple Inheritance:

When derived class inherits features from more than one base class then it is called Multiple Inheritance.

Syntax:



```
class Baseclass1:

        <body of base class1>

class Baseclass2:

        <body of base class2>

class Derivedclass(Baseclass1,Baseclass2):

        <body of the derived class>
```

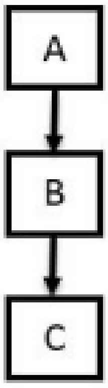
e.g.

```
class A:
        i=10
class B:
        j=20
class C(A,B):
        k=30
obj=C()
print("member of class A is",obj.i)
print("member of class B is",obj.j)
print("member of class C is",obj.k)
```

Multi-Level Inheritance:

When derived class inherits features from other derived classes then it is called Multi-level inheritance.

Syntax:



```
class Baseclass:

        <body of base class>

class Derivedclass1(Baseclass):

        <body of derived class 1>

class Derivedclass2(Derivedclass1):
        <body of the derived class2>
```

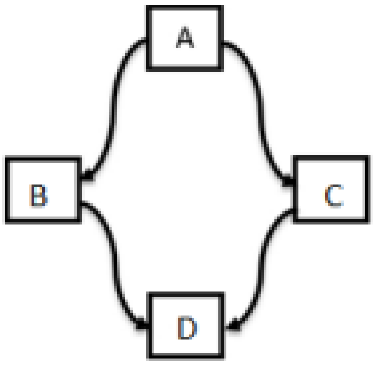
e.g.

```
class A:
        i=10
class B(A):
        j=20
class C(B):
        k=30
obj=C()
print("member of class A is",obj.i)
print("member of class B is",obj.j)
print("member of class C is",obj.k)
```

Multi Path Inheritance:

Syntax:

```
class Baseclass:
        <body of the base class>
class Derived1(Baseclass):
        <body of the derived1>
class Derived2(Baseclass):
        <body of the derived2>
class Derived3 (Derived1,Derived2) :
        <body of derived3>
```



e.g.

```
class A:
        i=10
class B(A):
        j=20
class C(A):
        k=30
class D(B,C):
        ijk=40
obj=D()
print("member of class A is",obj.i)
print("member of class B is",obj.j)
print("member of class C is",obj.k)
print("member of class Cis",obj.ijk)
```

Polymorphism:

The word polymorphism means having many forms. In python we can find the same operator or function taking multiple forms. That helps in re using a lot of code and decreases code complexity.

Polymorphism in operators

- The + operator can take two inputs and give us the result depending on what the inputs are.
- In the below examples we can see how the integer inputs yield an integer and if one of the input is float then the result becomes a float. Also for strings, they simply get concatenated.

Example:

```
a = 23
b = 11
c = 9.5
s1 = "Hello"
s2 = "There!"
print(a + b)
print(type(a + b))
print(b + c)
print(type (b + c))
print(s1 + s2)
print(type(s1 + s2))
```

Polymorphism in built-in functions

We can also see that different python functions can take inputs of different types and then process them differently. When we supply a string value to len() it counts every letter in it. But if we give tuple or a dictionary as an input, it processes them differently.

Example:

```
str = 'Hi There !'
tup = ('Mon','Tue','wed','Thu','Fri')
lst = ['Jan','Feb','Mar','Apr']
dict = {'1D':'Line','2D':'Triangle','3D':'Sphere'}
print(len(str))
print(len(tup))
print(len(lst))
print(len(dict))
```

Polymorphism in inheritance:

Method Overriding:

It is nothing but same method name in parent and child class with different functionalities. In inheritance only we can achieve method overriding. If super and sub classes have the same method name and if we call the overridden method then the method of corresponding class (by using which object we are calling the method) will be executed.

e.g.

```
class A:
        i=10
        def display(self):
                print("I am class A and I have data",self.i)
class B(A):
        j=20
        def display(self):
                print("I am class B and I have data",self.j)
obj=B()
obj.display()
```

```
OUTPUT :
I am class B and I have data 20
```

Note: In above program the method of class B will execute. If we want to execute method of class A by using Class B object we use super() concept.

Super():

In method overriding , If we want to access super class member by using sub class object we use super()

e.g

```
class A:
        i=10
        def display(self):
                print("I am class A and I hava data",self.i)
class B(A):
        j=20
        def display(self):
                super().display()
                print("I am class B and I hava data",self.j)
obj=B()
obj.display()
```

OUTPUT:

```
I am class A and I have  data 10
I am class B and I have  data 20
```

Note: In above example both the functions (display () in class A and display () in class B) will execute

Note: Name mangling is the encoding of function and variable names into unique names so that linkers can separate common names in the language.

<u>overloading operators</u>

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example, operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

```
# Python program to show use of + and * operator for different purposes.

print(1 + 2)
# concatenate two strings
print("Learn"+"For")
# Product two numbers
print(3 * 4)
# Repeat the String
print("Learn"*4)
```

Output:

```
3
LearnFor
12
LearnLearnLearnLearn
```

Example 2:

Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class and operators work according to that behavior defined in methods. When we use + operator, the magic method __add__ is automatically invoked in which the operation for + operator is defined.

```
class A:
        def  __init__(self, a):
                self.a = a
        def  __add__(self, o):                    # adding two objects
                return  self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("sai")
ob4 = A("kumar")
ob5=A([2,5,6,2])
ob6=A([34.6,12])
print(ob1 + ob2)
print(ob3 + ob4)
print(ob5 + ob6)
```

```
Ob1.a=1
Ob2.a=2
Ob3.a=”sai”
Ob4.a=”kumar”
Ob5.a=[ 2,5,6,2    ]
Ob6.a=[  34.6,12 ]
```

OUTPUT:

```
>>>
3
saikumar
[2, 5, 6, 2, 34.6, 12]
```

Case Study An ATM:

```
class ATM:
        def __init__(self):
                self.balance=0
                print("new account created")
        def deposit(self):
                amount=int(input("enter amount to deposit"))
                self.balance=self.balance+amount
                print("new balance is:",self.balance)
        def withdraw(self):
                amount=int(input("enter amount to withdraw"))
                if self.balance<amount:
                        print("Insufficient Balance")
                else:
                        self.balance=self.balance-amount
                        print("new balance is:",self.balance)
        def enquiry(self):
                print("Balance is:",self.balance)

a=ATM()
a.deposit()
a.withdraw()
a.enquiry()
```

OUTPUT:

```
>>>
new account created
enter amount to deposit15000
new balance is: 15000
enter amount to withdraw5648
new balance is: 9352
Balance is: 9352
```

<u>Adding and retrieving dynamic attributes of classes:</u>

Dynamic attributes in Python are terminologies for attributes that are defined at runtime, after creating the objects or instances.

<u>Example:</u>

```
class EMP:
        employee = True
e1 = EMP()
e2 = EMP()
e1.employee = False
e2.name = "SAI KUMAR"    #DYNAMIC ATTRIBUTE
print(e1.employee)
print(e2.employee)
print(e2.name)
print(e1.name)          # this will raise an error  as name is a dynamic attribute created only for
                        #the e2 object
```