Day-6: OOPS and EXCEPTION

28/06/2018

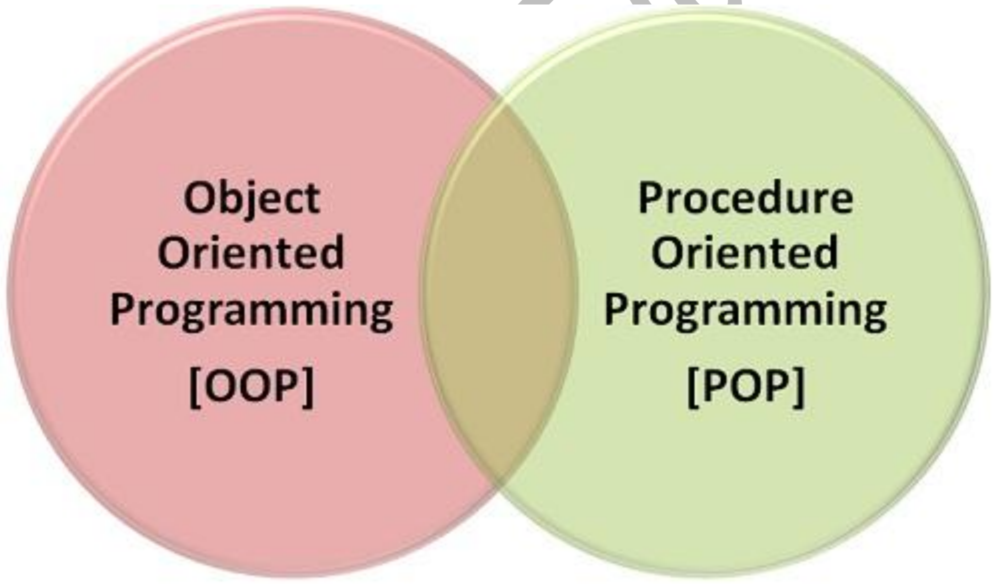
Workshop on “Introduction to Python”

Department Of Computer Science Engineering

SKFGI, Mankundu

Different Programming Models

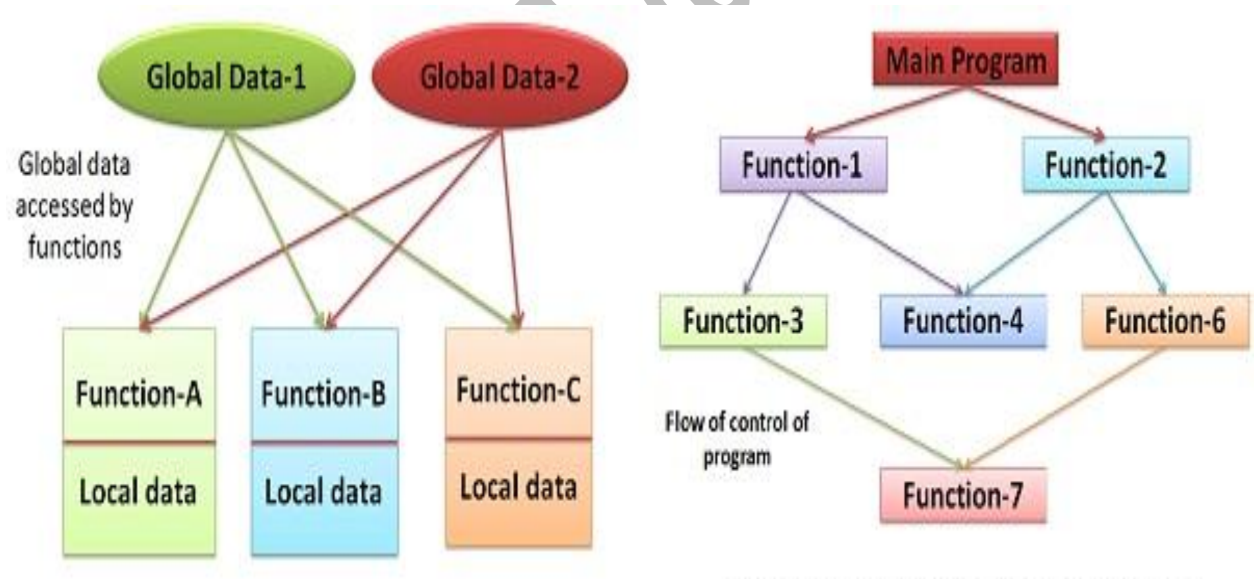
Both are programming processes whereas OOP stands for "Object Oriented Programming" and POP stands for "Procedure Oriented Programming". Both are programming languages that use high-level programming to solve a problem but using different approaches. These approaches in technical terms are known as programming paradigms. A programmer can take different approaches to write a program because there's no direct approach to solve a particular problem. This is where programming languages come to the picture. A program makes it easy to resolve the problem using just the right approach or you can say 'paradigm'. Object-oriented programming and procedure-oriented programming are two such paradigms.


Object Oriented Programming [OOP]
Procedure Oriented Programming [POP]

What is Procedure Oriented Programming (POP)?

POP follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions. Each step is carried out in order in a systematic manner so that a computer can understand what to do. The program is divided into small parts called functions and then it follows a series of computational steps to be carried out in order.

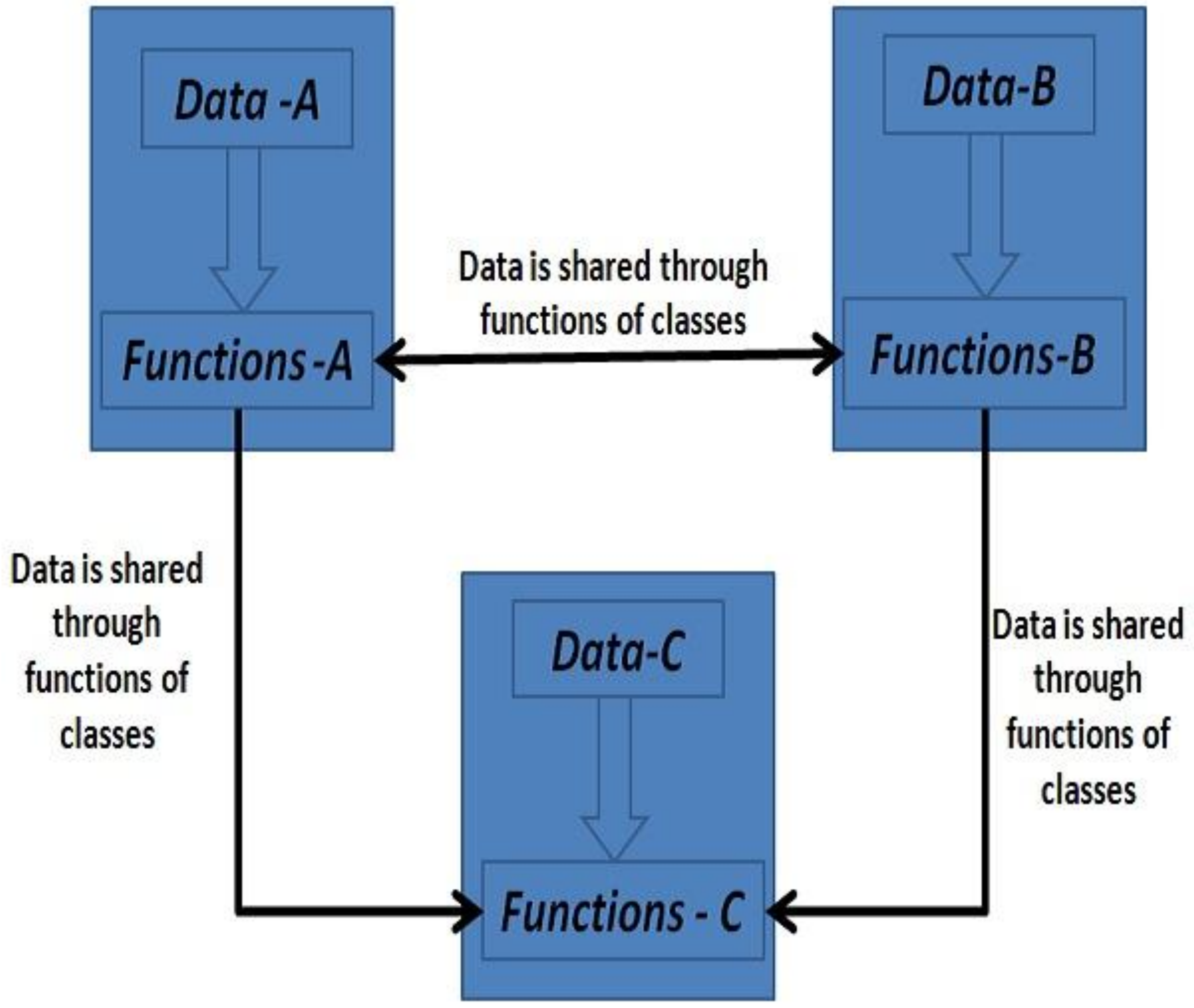
It follows a top-down approach to actually solve a problem, hence the name. Procedures correspond to functions and each function has its own purpose. Dividing the program into functions is the key to procedural programming. So a number of different functions are written in order to accomplish the tasks.



Structure of procedure oriented program

What is Object Oriented Programming (OOP)?

OOP is a high-level programming language where a program is divided into small chunks called objects using the object-oriented model, hence the name. This paradigm is based on objects and classes.


Data -A
Functions -A
Data-B
Functions-B
Data is shared through functions of classes
Data is shared through functions of classes
Data-C
Functions - C
Data is shared through functions of classes

Data flow in object-oriented programming

Difference Between POP between OOP:-

BASIS FOR COMPARISON	POP	OOP
Basic	Procedure/Structure oriented .	Object oriented.
Approach	Top-down.	Bottom-up.
Division	Large program is divided into units called functions.	Entire program is divided into objects.
Basis	Main focus is on "how to get the task done" i.e. on the procedure or structure of a program .	Main focus is on 'data security'. Hence, only objects are permitted to access the entities of a class.
Entity accessing mode	No access specifier observed.	Access specifier are "public", "private", "protected".
Overloading/Polymorphism	Neither it overload functions nor	It overloads functions, constructors, and

	operators.	operators.
Inheritance	Their is no provision of inheritance.	Inheritance achieved in three modes public private and protected.
Data hiding & security	There is no proper way of hiding the data, so data is insecure	Data is hidden in three modes public, private, and protected. hence data security increases.
Data sharing	Global data is shared among the functions in the program.	Data is shared among the objects through the member functions.
Example	C, VB, FORTRAN, Pascal	C++, JAVA, VB.NET, C#.NET.

Object Oriented Programming

Python is an object-oriented programming language. It allows us to develop applications using Object Oriented approach. In Python, we can easily create and use classes and objects.

Major principles of object-oriented programming system are given below

1. Class
2. Object
3. Inheritance
4. Polymorphism
5. Encapsulation
6. Data Abstraction

1.1What is a class?

A class is a blueprint for the object. Or A class is a code template for creating objects. Objects have member variables and have behaviour associated with them. In python a class is created by the keyword class.

Syntax:-

```
class <class_name>:
<statement-1>
<statement-2>

<statement-N>
```

1.1.1 Python class keyword

The user defined objects are created using the class keyword. The class is a blueprint that defines a nature of a future object. From classes we construct instances. An *instance* is a specific object created from a particular class. For example, Huck might be an instance of a Dog class.

class First():
   pass
fr = First()
print(type(fr))
print(type(First))
	<class '__main__.First'>
	<class 'type'>

Here we create a new instance of the First class. Or in other words, we instantiate the First class. The fr is a reference to our new object. Here we see that fr is an instance object of the First class.

Inside a class, we can define attributes and methods.

1.2 What is an Object?

Creating an object or an instance of a class is known as class instantiation.Object is an entity that has state and behavior. It may be anything. For example: mouse, keyboard, chair, table, pen etc. Everything in Python is an object.

Syntax:-

object_name=class name()

Accessing Variables Through Object

Prog1:-

```
        class MyClass:
                    v = 7
        obj = MyClass()
        print(obj.v)
```

prog2:-

```
    class MyClass1:
            v1 = "supreme"
    obj1 = MyClass1()
    print(obj1.v1)
```

Prog3:-

```
    class MyClass2:
            v= 7
    obj = MyClass2()
    obj1 = MyClass2()
    print(obj.v)
    print(obj1.v)
```

Prog4:-

```
        class MyClass3:
                 var = 7
        obj = MyClass3()
        obj1 = MyClass3()
        print(obj.var)
        print(obj1.var)
        obj1.var=9
        print(obj.var)
        print(obj1.var)
```

1.3.Methods

A ***method*** defines operations that we can perform with our objects.Method is a function that is associated with an object. When you define methods, you will need to always provide the first argument to the method with a self keyword.The self argument refers to the object itself. That is, the object that has called the method. This means that even if a method that takes no arguments, it should be defined to accept the self. Similarly, a function defined to accept one parameter will actually take two-self and the parameter.

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are essential in the *encapsulation* concept of the OOP paradigm. For example, we might have a connect() method in our AccessDatabase class. We need not to be informed how exactly the method connect connects to the database. We only know that it is used to connect to a database. This is essential in dividing responsibilities in programming, especially in large applications.

The method differs from a function only in two aspects:

- it belongs to a class and it is defined within a class
- the first parameter in the definition of a method has to be a reference "self" to the instance of the class
- a method is called without this parameter "self"

Example:-

```
class Circle():
    pi = 3.141592
    def __init__(self, radius=1):
        self.radius = radius
    def area(self):
        return self.radius * self.radius * Circle.pi
    def setRadius(self, radius):
        self.radius = radius
    def getRadius(self):
        return self.radius
c = Circle(5)
print(c.getRadius())
print(c.area())
c.setRadius(1)
print(c.getRadius())
print(c.area())
```

In the code example, we have a Circle class. We define three new methods.

The area() method returns the area of a circle.

The setRadius() method sets a new value for the radius attribute.

The getRadius() method returns the current radius.

```
c.setRadius(1)
```

The method is called on an instance object. The c object is paired with the self parameter of the class definition. The number 1 is paired with the radius parameter.

In Python, we can call methods in two ways. There are **bounded** and **unbounded** method calls.

Example:-

```
class Methods():
    def __init__(self):
        self.name = 'Methods'
    def getName(self):
        return self.name
m = Methods()
print(m.getName())
print(Methods.getName(m))
```

In this example, we demostrate both method calls.
print(m.getName())
This is the *bounded* method call. The Python interpreter automatically pairs the m instance with the self parameter.
print(Methods.getName(m))
And this is the *unbounded* method call. The instance object is explicitly given to the getName()method.

Prog1:-

```
class MyClass:
  def display(self):
      print("supreme")
obj = MyClass()
obj. display()
```

Prog2:-

```
class MyClass:
        def display1(self,v):
```

```
                    print("This is a message inside the class.",v)
          obj = MyClass()
          obj. display1 (7)
```

1.3.1 Python special methods

Classes in Python programming language can implement certain operations with special method names. These methods are not called directly, but by a specific language syntax. This is similar to what is known as *operator overloading* in C++ or Ruby.

Example:-

```
class Book():
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return "Title:{0} , author:{1}, pages:{2} ".format(
            self.title, self.author, self.pages)
    def __len__(self):
        return self.pages
    def __del__(self):
        print("A book is destroyed")
book = Book("Inside Steve's Brain", "Leander Kahney", 304)
print(book)
print(len(book))
del book
```

In our code example, we have a book class. Here we introduce four special methods: __init__(), __str__(), __len__() and __del__().

```
book = Book("Inside Steve's Brain", "Leander Kahney", 304)
```

Here we call the __init__() method. The method creates a new instance of a Book class.

```
print(book)
```

The print keyword calls the __str__() method. This method should return an informal string representation of an object.

```
print(len(book))
```

The len() function invokes the __len__() method. In our case, we print the number of pages of our book.

```
del book
```

The del keyword deletes an object. It invokes its __del__() method.

1.4.The _init_() method(The class constructor):-

The _init_() method has a special significance in python classes.The __init__() method is automatically executed when an object of a class is created.The method is useful to initialize the variables of the class object.Note that the __init__()is prefixed as well as suffixed by double underscores.

Prog1:-

```
class Abc():
    def __init__(self):
        print("skf")
obj=Abc()
```

Prog2:-

```
class Addition:
     def __init__(self, a, b):
         self.s = a
         self.k = b
```

```
    def add(self):
        c=x.s+x.k
        print(c)
x = Addition(3,4.5)
x.add()
print(x.s,x.k)
```

1.5 Python object attributes

An ***attribute*** is a characteristic of an object. This can be for example a salary of an employee. Attributes are set in the __init__() method.

Example:-

```
class Cat():

    def __init__(self, name):
        self.name = name
missy = Cat('Missy')
lucky = Cat('Lucky')
print(missy.name)
print(lucky.name)
```

In this code example, we have a Cat class. The special method __init__() is called automatically right after the object has been created.
def __init__(self, name):

Each method in a class definition begins with a reference to the instance object. It is by convention named self. There is nothing special about the self name. We could name it this, for example. The second parameter, name, is the argument. The value is passed during the class initialization.

self.name = name

Here we pass an attribute to an instance object.

missy = Cat('Missy')

lucky = Cat('Lucky')

Here we create two objects: cats Missy and Lucky. The number of arguments must correspond to the __init__() method of the class definition. The 'Missy' and 'Lucky' strings become the nameparameter of the __init__() method.

```
print(missy.name)

print(lucky.name)
```

Here we print the attributes of the two cat objects. Each instance of a class can have their own attributes.

```
Missy
Lucky
```

The attributes can be assigned dynamically, not just during initialization. This is demonstrated by the next example.

Example:-

```
class Person():
    pass
p = Person()
p.age = 24
p.name = "Peter"
print("{0} is {1} years old".format(p.name, p.age))
```

We define and create an empty Person class.

```
p.age = 24
p.name = "Peter"
```

Here we create two attributes dynamically: age and name.

1.6 Python class attributes

So far, we have been talking about instance attributes. In Python there are also so called *class object attributes*. Class object attributes are same for all instances of a class.

Example:-

```
class Cat:
    species = 'mammal'
    def __init__(self, name, age):

        self.name = name
        self.age = age
missy = Cat('Missy', 3)
lucky = Cat('Lucky', 5)
print(missy.name, missy.age)
print(lucky.name, lucky.age)
print(Cat.species)
print(missy.__class__.species)
print(lucky.__class__.species)
```

In our example, we have two cats with specific name and age attributes. Both cats share some characteristics. Missy and Lucky are both mammals. This is reflected in a class level attribute species. The attribute is defined outside any method name in the body of a class.

```
print(Cat.species)
print(missy.__class__.species)
```

There are two ways how we can access the class object attributes: either via the name of the Catclass, or with the help of a special __class__ attribute.

1.7.Class Variable and Object Variable:-

There are two types of variables-class variables and object variables.

1.7.1Class variables are shared in the sense that they are accessed by all objects (instances) of that class. There is only copy of the class variable and when any one object makes a change to a class variable, the change is reflected in all the other instances as well.

1.7.2Object variables are owned by each individual object/instance of the class. In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance of the same class. An example will make this easy to understand.

Prog1:-

```
class ABC():
    a=0  #class varible
    def __init__(self,var):
        ABC.a+=1
        self.var=var #object variable
        print("The object value is:",var)
        print("The value of class variable is: ",ABC.a)
obj1=ABC(10)
obj2=ABC(20)
obj3=ABC(30)
```

1.8.Public and Private Data Members:-

Public variables are those variables that are defined in the class and can be accessed from anywhere in the program.

Private variables are those variables that are defined in the class with a double score prefix(__).These variables can be accessed only from within the class and from nowhere outside the class.

Example:-

```
class ABC():
    def __init__(self,var1,var2):
        self.var1=var1
        self.__var2=var2
    def display(self):
        print("From class method,var1=",self.var1)
        print("From class method,var2=",self.__var2)
obj=ABC(10,20)
obj.display()
print("From main module,var1=",obj.var1)
print("From main module,var2=",obj.__var2)
```

1.8.1Private Methods:-

A private method can be accessed using the object name as well as the class name from outside the class.

1.8.2 Public Method Example:-

```
class ABC():
    def display(self,var):
        print("From class method,var=",var)

obj=ABC()
obj.display(7)
```

Private Method Example:-

(Accessing approach is wrong)

```
class ABC():

    def __display(self,var):
        print("From class method,var=",var)
obj=ABC()
obj.__display(7)
```

CORRECT PROGRAM:-

```
class ABC():
    def __display(self,var):
        print("From class method,var=",var)
 obj=ABC()
 obj._ABC__display(7)
```

1.9. Inheritance:- Inheritance is a powerful feature in object oriented programming. Inheritance is a feature of object-oriented programming. It specifies that one object acquires all the properties and behaviors of parent object. By using inheritance you can define a new class with a little or no changes to the existing class. The new class is known as derived class or child class and from which it inherits the properties is called base class or parent class.

It provides re-usability of the code.

It refers to defining a new class with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

There are basically four types of Inheritance-

1.Single Inheritance

2. Multiple Inheritance

3. Multilevel Inheritance

4. Hybrid Inheritance

1.9.1Single Inheritance Syntax:-

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Prog1:-

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last
    def Name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        Person.__init__(self,first, last)
        self.staffnumber = staffnum
   def GetEmployee(self):
        return self.Name() + ", " +  self.staffnumber

x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")
print(x.Name())
print(y.GetEmployee())
```

1.7.2 Multiple Inheritance in Python

A class can be derived from more than one base classes in Python. This is called multiple inheritance.
In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

Syntax:-

```
class Base1:
    pass
class Base2:
    pass
class MultiDerived(Base1, Base2):
    pass
```

Prog1:-

```
class Person:
    #defining constructor
    def __init__(self, personName, personAge):
        self.name = personName
        self.age = personAge
      #defining class methods
    def showName(self):
        print(self.name)
    def showAge(self):
        print(self.age)
      #end of class definition

  # defining another class
class Student: # Person is the
    def __init__(self, studentId):
        self.studentId = studentId
    def getId(self):
        return self.studentId
class Resident(Person, Student): # extends both Person and Student class
    def __init__(self, name, age, id):
        Person.__init__(self, name, age)
        Student.__init__(self, id)
```

# Create an object of the subclass

```
resident1 = Resident('John', 30, '102')
resident1.showName()
print(resident1.getId())
```

1.7.3 Multilevel Inheritance

On the other hand, we can also inherit form a derived class. This is called multilevel inheritance.

In multilevel **inheritance**, features of the base class and the derived class is inherited into the new derived class.

SYNTAX:-

```
class Base:
    pass
class Derived1(Base):
    pass
class Derived2(Derived1):
    pass
```

Here, Derived1 is derived from Base, and Derived2 is derived from Derived1.

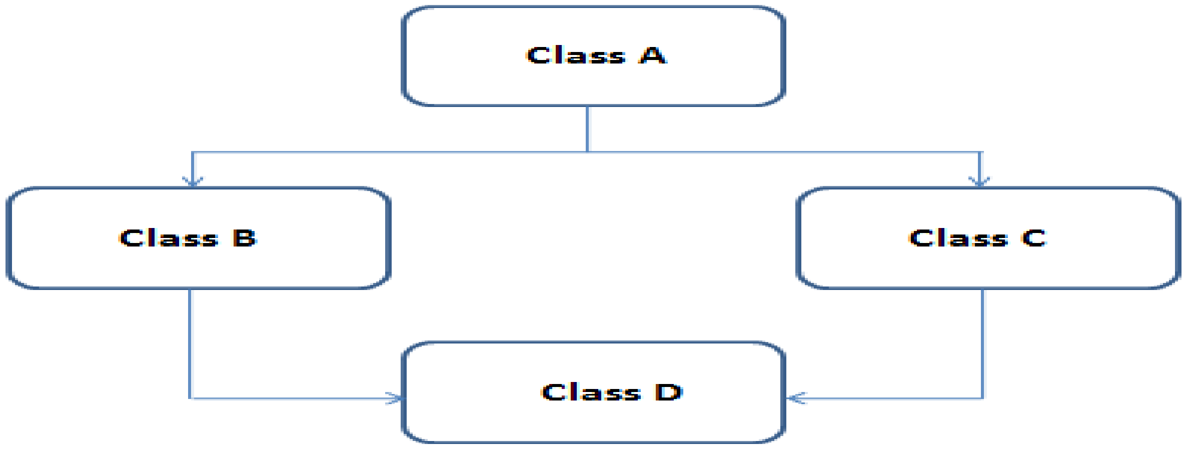
Prog1:-

```
class Animal:
    def eat(self):
      print ('Eating...')
class Dog(Animal):
   def bark(self):
      print ('Barking...')
class BabyDog(Dog):
    def weep(self):
```

```
    print ('Weeping...' )
d=BabyDog()
d.eat()
d.bark()
d.weep()
```

1.7.4 Multi-path Inheritance(Hybrid Inheritance)

Deriving a class from other derived classes that are in turn derived from the base class is called multi-path inheritance or hybrid inheritance.



1.8.Polymorphism

Polymorphism is made by two words "poly" and "morphs". Poly means many and Morphs means form, shape. It defines that one task can be performed in different ways. For example: You have a class animal and all animals talk. But they talk differently. Here, the "talk" behavior is polymorphic in the sense and totally depends on the animal. So, the abstract "animal" concept does not actually "talk", but specific animals (like dogs and cats) have a concrete implementation of the action "talk".

Example:-

```
a = "alfa"
b = (1, 2, 3, 4)
c = ['o', 'm', 'e', 'g', 'a']

print(a[2])
print(b[1])
print(c[3])
```

Python uses polymorphism extensively in built-in types. Here we use the same indexing operator for three different data types.

Example:-

```
class Parrot:

    def fly(self):
        print("Parrot can fly")
     def swim(self):
        print("Parrot can't swim")
class Penguin:
    def fly(self):
        print("Penguin can't fly")
    def swim(self):
        print("Penguin can swim")
# common interface
def flying_test(bird):
    bird.fly()
#instantiate objects
blu = Parrot()
peggy = Penguin()
# passing the object
flying_test(blu)
flying_test(peggy)
```

1.9.Method Overloading

In Python we can define a method in such a way that there are multiple ways to call it. Given a single method or function, we can specify the number of parameters ourself. Depending on the function definition, it can be called with zero, one, two or more parameters. This is known as *method* *overloading.* Not all programming languages support method overloading, but Python does.

<u>Prog1:-</u>

```
class Human:
    def sayHello(self, name=None):
        if name is not None:
            print ('WELCOME TO ' + name)
        else:
            print ('Hello ')
 # Create instance
obj = Human()
 # Call the method
obj.sayHello()
 # Call the method with a parameter
obj.sayHello('SUPREME')
```

1.10.Method Overriding

Override means having two methods with the same name but doing different tasks. If there is any method in the superclass and a method with the same name in a subclass, then by executing the method, the method of the corresponding class will be executed.

<u>Prog1:-</u>

```
class Rectangle():
        def __init__(self,length,breadth):
                self.length = length
                self.breadth = breadth
        def getArea(self):
                print (self.length*self.breadth," is area of rectangle")
class Square(Rectangle):
        def __init__(self,side):
                self.side = side
        def getArea(self):
                print (self.side*self.side," is area of square")
s = Square(4)
r = Rectangle(2,4)
s.getArea()
r.getArea()
```

1.11Encapsulation

Encapsulation is also the feature of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

Example:-

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price
c = Computer()
c.sell()
# change the price
c.__maxprice = 1000
c.sell()
# using setter function
c.setMaxPrice(1000)
c.sell()
```

In the above program, we defined a class *Computer*. We use __init__() method to store the maximum selling price of computer. We tried to modify the price. However, we can't change it because Python treats the __*maxprice* as private attributes. To change the value, we used a setter function i.e setMaxPrice() which takes price as paramete

1.12Data Abstraction(Data Hiding)

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things, so that the name captures the core of what a function or a whole program does.

2.Exception

2.1.What is an Exception?

Exception can be said to be any abnormal condition in a program resulting to the disruption in the flow of the program.

Whenever an exception occurs the program halts the execution and thus further code is not executed. Thus exception is that error which python script is unable to tackle with.

Exception in a code can also be handled. In case it is not handled, then the code is not executed further and hence execution stops when exception occurs.

2.1.1Why use Exceptions?

Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then you can use exception handling.

3.Set up exception handling blocks

To use exception handling in Python, you first need to have a catch-all except clause. The words "try" and "except" are Python keywords and are used to catch exceptions.try-except [exception-name] blocks.The code within the try clause will be executed statement by statement.

If an exception occurs, the rest of the try block will be skipped and the except clause will be executed.

```
try:
    some statements here

except:

    exception handling
```

3.1.How does it work?

The error handling is done through the use of exceptions that are caught in try blocks and handled in except blocks. If an error is encountered, a try block code execution is stopped and transferred down to the except block.

In addition to using an except block after the try block, you can also use the finally block.

The code in the finally block will be executed regardless of whether an exception occurs.

3.2 List of Standard Exceptions –

EXCEPTION NAME AND DESCRIPTION	PROGRAM
Exception Base class for all exceptions	'2' + 2

ArithmeticError
Base class for all errors that occur for numeric calculation.
	try:
	    a = 10/0
	    print(a)
	except ArithmeticError:
	        print ("This statement is raising an arithmetic exception.")
	else:
	    print ("Success.")

OverflowError
Raised when a calculation exceeds maximum limit for a numeric type.
	i=1
	try:
	    f = 3.0**i
	    for i in range(100):
	        print (i, f)
	        f = f ** 2
	except OverflowError as err:
	    print ('Overflowed after ', f, err)

ZeroDivisionError
Raised when division or modulo by zero takes place for all numeric types.
	i=int(input("enter a number"))
	
	j=int(input("enter a number"))
	
	try:
	
	       k=i/j
	
	       print(j)
	
	except ZeroDivisionError:
	
	        print('divided by zero')

AssertionError
Raised in case of failure of the Assert statement.
	c=int(input("enter a number"))
	
	assert(c>=18), "u can not cast your vote"
	
	print(c)

AttributeError
Raised in case of failure
	x = [1, 2, 3]
	x.next()

of attribute reference or assignment.	x.next() x.next() x.next()
ImportError Raised when an import statement fails.	math.sqrt(5,2)

KeyboardInterrupt
Raised when the user interrupts program execution, usually by pressing Ctrl+c.
	i=int(input("enter a number"))
	j=int(input("enter a number"))
	try:
	        k=i/j
	        print(j)
	except ZeroDivisionError:
	         print('divided by zero')
	except KeyboardInterrupt:
	         print('keyboardinterrupt')

IndexError
Raised when an index is not found in a sequence.
	l=[0,1,1,2]
	print(l[5])

KeyError Raised when the specified key is not found in the dictionary.	>>>a={'name':'sayon','age':35} >>>a[‘name’] >>>a[name1’]

NameError
Raised when an identifier is not found in the local or global namespace.
	anotherList = [4, 3, 5, 8, 1]
	
	anather.append(4)

UnboundLocalError
Raised when trying to access a local variable in a function or method but no value has been assigned to it.
	def f():
	    x=x*2
	    print(x)
	
	f()

IOError
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.
	name = "nope.txt"
	with open(name) as f:
	    print(f.readline())

SyntaxError	print "hi"

Raised when there is an error in Python syntax.	
IndentationError Raised when indentation is not specified properly.	
TypeError Raised when an operation or function is attempted that is invalid for the specified data type.	10/'a'
ValueError Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.	num = int(input("Enter the number ")) re = 100/num

EXCEPTION NAME AND DESCRIPTION	**PROGRAM**
StopIteration Raised when the next() method of an iterator does not point to any object.	
SystemExit Raised by the sys.exit() function.	
StandardError Base class for all built-in exceptions except StopIteration and SystemExit.	
FloatingPointError Raised when a floating point calculation fails.	
EOFError Raised when there is no input from either the input() function and the end of file is reached.	
LookupError Base class for all lookup	

errors.	
EnvironmentError Base class for all exceptions that occur outside the Python environment.	
SystemError Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.	
SystemExit Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.	
RuntimeError Raised when a generated error does not fall into any category.	
NotImplementedError Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented	

4.Multiple Except Blocks

Python allows to have multiple except blocks for a single try block.The block which matches with the exception generated will get executed.

Prog1:-

class ExFile:

```
try:
   num = int(input("Enter the number "))
   re = 100/num
except ValueError :
   print ("Value is not int type")
except ZeroDivisionError :
   print ("Don't use zero" )
else:
   print ("result is ",re)
```

5.Try ... except ... else clause:

The else clause in a try , except statement must follow all except clauses, and is useful for code that must be executed if the try clause does not raise an exception.

```
try:
    data = something_that_can_go_wrong
except IOError:
    handle_the_exception_error
else:
    doing_different_exception_handling
```

Exceptions in the else clause are not handled by the preceding except clauses.

Make sure that the else clause is run before the finally block.

Prog1:-

```
class ExFile:
    i= int(input("Enter the number"))
    j=int(input("Enter the number"))
    try:
        k=i/j
    except ZeroDivisionError:
        print('divided by zero')
    else:
            print('number is not divided by zero')
```

6. Try … finally clause

The finally clause is optional. It is intended to define clean-up actions that must be executed under all circumstances

A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.

```
class ExFile1:
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print ("Error: can\'t find file or read data")
```

7.Raising an Exception(User Defined Exception)

We can raise an exception in our own program by using the raise exception statement.Raising an exception breaks current code execution and returns the exception back until it is handled.

```
try:
    n = 7
    print(n)
    raise ValueError
except:
    print("Exception occurred")
```