Unit -II
Object Oriented Programming

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy.

Difference between procedure oriented and object oriented programming:-

S.no.	On the basis of	Procedural Programming	Object-oriented programming
1.	Definition	It is a programming language that is derived from structure programming and based upon the concept of calling procedures. It follows a step-by-step approach in order to break down a task into a set of variables and routines via a sequence of instructions.	Object-oriented programming is a computer programming design philosophy or methodology that organizes/ models software design around data or objects rather than functions and logic.
2.	Security	It is less secure than OOPs.	Data hiding is possible in object-oriented programming due to abstraction. So, it is more secure than procedural programming.
3.	Approach	It follows a top-down approach.	It follows a bottom-up approach.
4.	Data movement	In procedural programming, data moves freely within the system from one function to another.	In OOP, objects can move and communicate with each other via member functions.
5.	Orientation	It is structure/procedure-oriented.	It is object-oriented.
6.	Access modifiers	There are no access modifiers in procedural programming.	The access modifiers in OOP are named as private, public, and protected.
7.	Inheritance	Procedural programming does not have the concept of inheritance.	There is a feature of inheritance in object-oriented programming.
8.	Code reusability	There is no code reusability present in procedural programming.	It offers code reusability by using the feature of inheritance.
9.	Overloading	Overloading is not possible in procedural programming.	In OOP, there is a concept of function overloading and operator overloading.

10.	**Importance**	It gives importance to functions over data.	It gives importance to data over functions.
11.	**Virtual class**	In procedural programming, there are no virtual classes.	In OOP, there is an appearance of virtual classes in inheritance.
12.	**Complex problems**	It is not appropriate for complex problems.	It is appropriate for complex problems.
13.	**Data hiding**	There is not any proper way for data hiding.	There is a possibility of data hiding.
14.	**Program division**	In Procedural programming, a program is divided into small programs that are referred to as functions.	In OOP, a program is divided into small parts that are referred to as objects.
15.	**Examples**	Examples of Procedural programming include C, Fortran, Pascal, and VB.	The examples of object-oriented programming are - .NET, C#, Python, Java, VB.NET, and C++.

So, that's all about the article. Hope the article is informative and interesting to you, and you gain knowledge about procedural programming, object-oriented programming, and their comparison.

Features of oop:-

1)**Python Classes:-**

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Example

Create a class named MyClass, with a property named x:

```
Class MyClass:
  x = 5
```

2) Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.To understand the meaning of classes we have to understand the built-in __init__() function.All classes have a function called __init__(), which is always executed when the class is being initiated.

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

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class  Person:
 def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

Note: The __init__() function is called automatically every time the class is being used to create a new object.

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
 def __init__(self, name, age):
    self.name = name
    self.age = age
 def myfunc(self):
    print("Hello my name is " + self.name)
    print("Hello my age is " + self.age)
p1 = Person("John", 36)
```

```
p1.myfunc()
```

Note: The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words mysillyobject and abc instead of self:

```
class Person:
 def __init__(mysillyobject, name, age):
   mysillyobject.name = name
   mysillyobject.age = age
 def myfunc(abc):
   print("Hello my name is " + abc.name)
p1 = Person("John", 36)
p1.myfunc()
```

Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the del keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the del keyword:

Example

Delete the p1 object:

```
del p1
```

Example:-

```
class Employee:
 def __init__(self, name, salary):
  self.name = name
  self.salary= salary
 def display Employee(self):
  print"Name : ",self.name,", Salary: ",self.salary
emp1 =Employee("madhvi",2000)
emp2 =Employee("Mayuri",5000)
emp1.displayEmployee()
emp2.displayEmployee()
```

Output:-

Name : madhvi , Salary: 2000

Name : Mayuri , Salary: 5000

Example:-

```
class Student:
 def __init__(self, rollno, name):
      self.rollno = rollno
      self.name = name
 def displayStudent(self):
      print "rollno : ", self.rollno,  ", name: ",
       self.name
emp1 = Student(121, "Ajeet")
emp2 = Student(122, "Sonam")
emp1.displayStudent()
emp2.displayStudent()
```

3) Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Creating Methods in Python

```
class Parrot:
       # instance attributes
  def __init__(self, name, age):
       self.name = name
       self.age = age
       # instance method
   def sing(self, song):
      return "{} sings {}".format(self.name, song)
     def dance(self):
     return "{} is now dancing".format(self.name)
```

```
# instantiate the object
blu = Parrot("Blu", 10)
# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Output

```
Blu sings 'Happy'
Blu is now dancing
```

4) Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example: Use of Inheritance in Python

```
# parent class
class Bird:
   def __init__(self):
    print("Bird is ready")
  def whoisThis(self):
    print("Bird")
  def swim(self):
    print("Swim faster")
class Penguin(Bird):
  def __init__(self):
        # call super() function
    super().__init__()
    print("Penguin is ready")
  def whoisThis(self):
   print("Penguin")
  def run(self):
  print("Run faster")
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Output

```
Bird is ready
```

Penguin is ready
Penguin
Swim faster
Run faster

5) Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single _ or double __.

Example : Data Encapsulation in Python

```
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()
```

Output

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

6) Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

Example: Using Polymorphism in Python

```
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)
```

Constructors:-

A constructor is a special type of method (function) which is used to initialize the instance members of the class. Constructor can be parameterized and non-parameterized as well. Constructor definition executes when we create object of the class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating a Constructor

A constructor is a class function that begins with double underscore (_). The name of the constructor is always the same __init__().

While creating an object, a constructor can accept arguments if necessary. When we 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.

Constructors can be parameterized and non-parameterized as well. The parameterized constructors are used to set custom value for instance variables that can be used further in the application.

Non Parameterized Constructor Example

```
class Student:
    def __init__(self):
        print("This is non parametrized constructor")
    def show(self,name):
        print("Hello",name)
student = Student()
student.show("Abhi")
```

Output:

```
This is non parametrized constructor
Hello Abhi
```

Parameterized Constructor Example

```
class Student:
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
student = Student("Sani")
student.show()
```

Output:

```
This is parametrized constructor
Hello Sani
```

Variables

Variable is a name which is used to refer memory location. Variable also known as identifier and used to hold value.

In Python, we don't need to specify the type of variable because Python is a type infer language and smart enough to get variable type.

Variable names can be a group of both letters and digits, but they have to begin with a letter or an underscore.

It is recomended to use lowercase letters for variable name. Rahul and rahul both are two different variables.

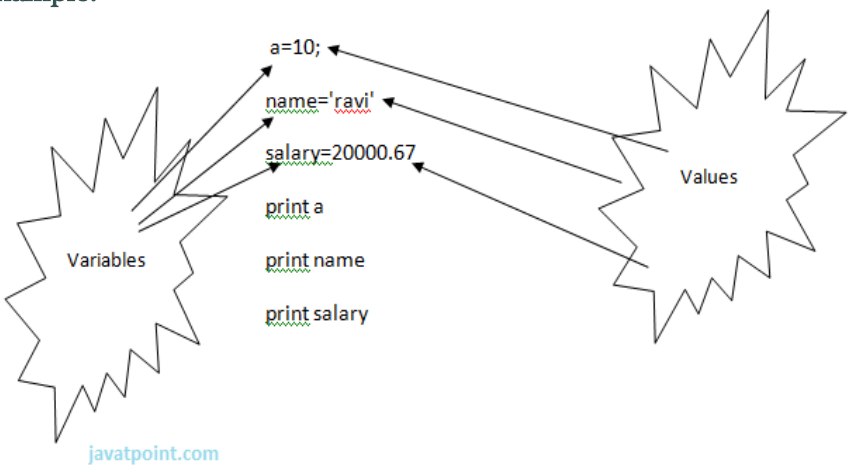
Declaring Variable and Assigning Values

Python does not bound us to declare variable before using in the application. It allows us to create variable at required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

Example:-



Output:

```
10
```

ravi

20000.67

Example:-

```
x = 5
y = "John"
print(x)
print(y)
```

Variable Names

1)A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:
2)A variable name must start with a letter or the underscore character
3)A variable name cannot start with a number
4)A variable name can only contain alpha-numeric characters and underscores (A-Z, 0-9, and _,a-z)
5)Variable names are case-sensitive (age, Age and AGE are three different variables)

Multiple Assignment

Python allows us to assign a value to multiple variables in a single statement which is also known as multiple assignment.
We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Lets see given examples.

1. Assigning single value to multiple variables

Eg:

```
x=y=z=50
print x
print y
print z
```

Output:

50

50

50

2.Assigning multiple values to multiple variables:

Example:-

```
a,b,c=5,10,15
print a
print b
print c
```

Output:

5

10

The values will be assigned in the order in which variables appears.

Namespaces:-

Generally speaking, a namespace(sometimes also called a context) is a naming system for making names unique to avoid ambiguity. Everybody knows a namespacing system from daily life, i.e. the naming of people in firstname and familiy name (surname).

Another example is a network: each network device (workstation, server, printer, ...) needs a unique name and address.

Yet another example is the directory structure of file systems. The same file name can be used in different directories, the files can be uniquely accessed via the pathnames. Many programming languages use namespaces or contexts for identifiers.

An identifier defined in a namespace is associated with that namespace. This way, the same identifier can be independently defined in multiple namespaces. (Like the same file names in different directories) Programming languages, which support namespaces, may have different rules that determine to which namespace an identifier belongs.

Namespaces in Python are implemented as Python dictionaries, this means it is a mapping from names (keys) to objects (values). The user doesn't have to know this to write a Python program and when using namespaces.

Some namespaces in Python:

- global names of a module
- local names in a function or method invocation
- built-in names: this namespace contains built-in functions (e.g. abs(), cmp(), ...) and built-in exception names

For example, when we do the assignment a = 2, here 2 is an object stored in memory and a is the name we associate it with. We can get the address (in RAM) of some object through the built-in function, id().

Example:-

```
a = 2
print('id(2) =', id(2))
print('id(a) =', id(a))
```

Example:-

```
a = 2
print('id(a) =', id(a))
a = a+1
print('id(a) =', id(a))
print('id(3) =', id(3))
b = 2
print('id(2) =', id(2))
```

Types of methods:-

There are three methods.

Instance Methods

Instance methods are the most common type of methods in Python classes. These are so called because they can access unique data of their instance. If you have two objects each created from a car class, then they each may have different properties. They may have different colors, engine sizes, seats, and so on.

Instance methods must have **self** as a parameter, but you don't need to pass this in every time. Self is another Python special term. Inside any instance method, you can use self to access any data or methods that may reside in your class. You won't be able to access them without going through self.

Example:-

```
class DecoratorExample:
  def __init__(self):
    print('Hello, World!')
    self.name = 'Decorator_Example'
def example_function(self):
    print('I\'m an instance method!')
    print('My name is ' + self.name)
de = DecoratorExample()
de.example_function()
```

Class Methods

Class methods are the third and final OOP method type to know. Class methods know about their class. They can't access specific instance data, but they can call other static methods. Class methods don't need **self** as an argument, but they do need a parameter called **cls**. This stands for **class**, and like self, gets automatically passed in by Python.

Class methods are created using the **@classmethod** decorator.

Example:-

```
class DecoratorExample:
  def __init__(self):
    print('Hello, World!')
   @classmethod
  def example_function(cls):
    print('I\'m a class method!')
   cls.some_other_function()
   @staticmethod
  def some_other_function():
    print('Hello!')

de = DecoratorExample()
de.example_function()
```

Output:-

```
Hello, World!
I'm a class method!
Hello!
```

Example:-

```
from datetime import date
class Person:
    def __init__(self, name, age):
        self.name = name
self.age = age
    @classmethod
deffromBirthYear(cls, name, birthYear):
        return cls(name, date.today().year - birthYear)
    def display(self):
        print(self.name + "'s age is: " + str(self.age))
person = Person('Amar', 19)
person.display()
person1 = Person.fromBirthYear('John',  1985)
person1.display()
```

Example:-

```
class Pizza:
    def __init__(self, ingredients):
     self.ingredients = ingredients
   def __repr__(self):
        return f'Pizza({self.ingredients!r})'
    @classmethod
def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])
    @classmethod
def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])
```

Static Methods

Static methods are methods that are related to a class in some way, but don't need to access any class-specific data. You don't have to use **self**, and you don't even need to instantiate an instance, you can simply call your method:
Therefore a static method can neither modify object state nor class state. Static methods are restricted in what data they can access - and they're primarily a way to namespace your methods.

Example:-

```
class Calculator:
    @staticmethod
 def multiplyNums(x, y):
        return x + y
print('Product:', Calculator.multiplyNums(15, 110))
```

Output:-

```
125
```

Example:-

```
class DecoratorExample:
  def __init__(self):
    print('Hello, World!')
  @staticmethod
  def example_function():
    print('I\'m a static method!')
de = DecoratorExample.example_function()
```

Output:-

```
I'm a static method!
```

Inner class:-

A class defined in another class is known as an inner class or nested class. If an object is created using child class means inner class then the object can also be used by parent class or root class. A parent class can have one or more inner classes but generally inner classes are avoided. We can make our code even more object-oriented by using an inner class. A single object of the class can hold multiple sub-objects. We can use multiple sub-objects to give a good structure to our program.

Example:

- First, we create a class and then the constructor of the class.
- After creating a class, we will create another class within that class, the class inside another class will be called an inner class.

```
class Color:
    def __init__(self):
        self.name = 'Green'
        self.lg = self.Lightgreen()
    def show(self):
        print ('Name:', self.name)
class Lightgreen:
    def __init__(self):
        self.name = 'Light Green'
        self.code = '024avc'
    def display(self):
        print ('Name:', self.name)
        print ('Code:', self.code)
    outer = Color()
outer.show()
g = outer.lg
g.display()
```

What is Inheritance

Inheritance is a feature of Object Oriented Programming. It is used to specify that one class will get most or all of its features from its parent class. It is a very powerful feature which facilitates users to create a new class with a few or more modification to an existing class. The new class is called child class or derived class and the main class from which it inherits the properties is called base class or parent class.The child class or derived class inherits the features from the parent class, adding new features to it. It facilitates re-usability of code.Image representation:

Syntax

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    <statement-N>
```

Example

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

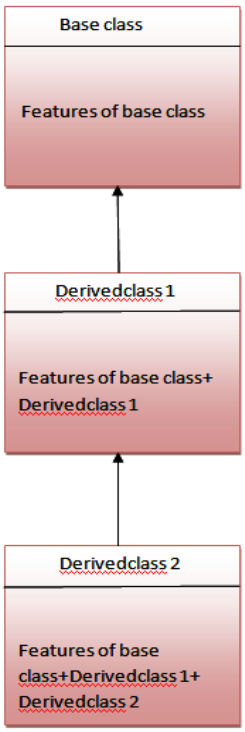
Output:

1. Eating...
2. Barking...

Multilevel Inheritance

Multilevel inheritance is also possible in Python like other Object Oriented programming languages. We can inherit a derived class from another derived class, this process is known as multilevel inheritance. In Python, multilevel inheritance can be done at any depth.

Image representation:



Example:-

```
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()
```

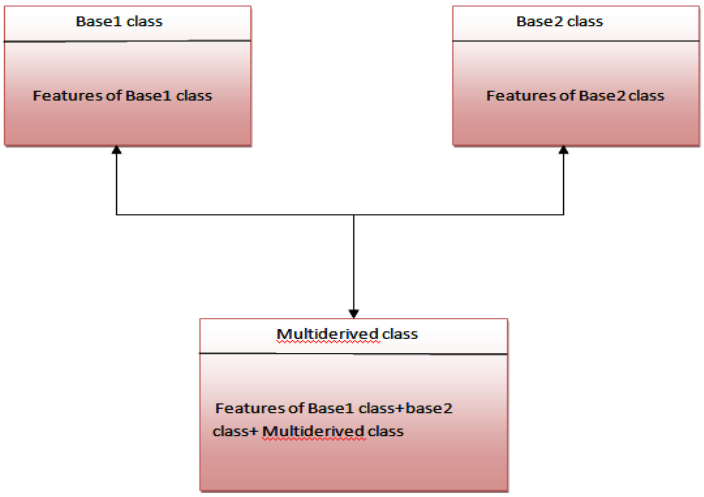
Output:

```
Eating...
Barking...
Weeping
```

Multiple Inheritance

Python supports multiple inheritance too. It allows us to inherit multiple parent classes. We can derive a child class from more than one base (parent) classes.

Image Representation



The multiderived class inherits the properties of both classes base1 and base2.

Syntax:-

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Or

```
class Base1:
    pass
 class Base2:
    pass
```

Example:-

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

Python Multiple Inheritance Example

Example:-

```
class First(object):
  def __init__(self):
    super(First, self).__init__()
    print("first")
class Second(object):
  def __init__(self):
    super(Second, self).__init__()
    print("second")
class Third(Second, First):
  def __init__(self):
    super(Third, self).__init__()
```

```
    print("third")
 Third();
```

Output:
first
second
third

super () keyword

The super() method is most commonly used with __init__ function in base class. This is usually the only place where we need to do some things in a child then complete the initialization in the parent.

Example:-

```
class Child(Parent):
    def __init__(self, stuff):
        self.stuff = stuff
        super(Child, self).__init__()
```

Example:-

```
class Base(object):
    def __init__(self):
        print "Base created"
class ChildA(Base):
    def __init__(self):
        Base.__init__(self)
class ChildB(Base):
    def __init__(self):
        super(ChildB, self).__init__()
ChildA()
ChildB()
```

Output:-

```
Base created
Base created
```

Example:-

```
class Mammal(object):
def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')
class Dog(Mammal):
  def __init__(self):
    print('Dog has four legs.')
    super().__init__('Dog')
d1 = Dog()
```

Output:-

Dog has four legs.

Dog is a warm-blooded animal.

Operator Overloading:-

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you. You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation:

Example

```
class Vector:
 def __init__(self, a, b):
  self.a = a
  self.b = b
 def __str__(self):
   return 'Vector (%d, %d)' % (self.a, self.b)
 def __add__(self,other):
   return Vector(self.a + other.a, self.b + other.b)
v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

Method Overloading

Like other languages do, python does not supports method overloading. We may overload the methods but can only use the latest defined method.

Example:-

```
def product(a, b):
    p = a * b
   print(p)
def product(a, b, c):
    p = a * b*c
    print(p)
product(4, 5, 5)
```

Output:

100

In the above code we have defined two product method, but we can only use the second product method, as **python does not supports method overloading**. We may define many **method of same name and different argument but we can only use the latest defined method.** Calling the other method will produce an error. Like here calling will produce an error as the latest defined product method takes three arguments.

However we may use other implementation in python to make the same function work differently i.e. as per the arguments.

Example:-

```
def add(datatype, *args):
    if datatype =='int':
       answer = 0
    if datatype =='str':
       answer =''
   for x in args:
       answer = answer + x
    print(answer)
add('int', 5, 6)
add('str', 'Hi ', 'Geeks')
```

Output:

```
11
Hi Geeks
```

Overriding Methods

You can always override your parent class methods. One reason for overridingparent's methods is because you may want special or different functionality in your subclass.

Example

```
class Parent:
 def myMethod(self):
   print 'Calling parent method'
class Child(Parent):
  def myMethod(self):
    print 'Calling child method'
c = Child()
c.myMethod()
```

Output:-

```
Calling child method
```

Example:-

```
class A:
 def sayhi(self):
         print("I'm in A")
class B(A):
 def sayhi(self):
         print("I'm in B")
bobj=B()
bobj.sayhi()
```

Abstract Classes:-

In object-oriented programming, an abstract class is a class that cannot be instantiated. However, you can create classes that inherit from an abstract class.
Typically, you use an abstract class to create a blueprint for other classes.
Similarly, an abstract method is an method without an implementation. An abstract class may or may not include abstract methods.
Python doesn't directly support abstract classes. But it does offer a module that allows you to define abstract classes.
To define an abstract class, you use the abc (abstract base class) module.
The abc module provides you with the infrastructure for defining abstract base classes.

Example:-

```
from abc import ABC, abstractmethod
class Car(ABC):
 def mileage(self):
 pass
class Tesla(Car):
 def mileage(self):
  print("The mileage is 30kmph")
class Suzuki(Car):
 def mileage(self):
  print("The mileage is 25kmph ")
class Duster(Car):
 def mileage(self):
  print("The mileage is 24kmph ")
class Renault(Car):
 def mileage(self):
   print("The mileage is 27kmph ")
 t= Tesla ()
t.mileage()
r = Renault()
r.mileage()
s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
```

Example:-

```
from abc import ABC, abstractmethod
class AbstractClassExample(ABC):
 def __init__(self, value):
 self.value = value
 super().__init__()
        @abstractmethod
def do_something(self):
 pass
class DoAdd42(AbstractClassExample):
```

```
 def do_something(self):
  return self.value + 42
class DoMul42(AbstractClassExample):
 def do_something(self):
 return self.value * 42
x = DoAdd42(10)
y = DoMul42(10)
print(x.do_something())
print(y.do_something())
```

Example:-

```
from abc import ABC, abstractmethod
class AbstractOperation(ABC):
 def __init__(self, operand_a, operand_b):
  self.operand_a = operand_a
  self.operand_b = operand_b
  super(AbstractOperation, self).__init__()
       @abstractmethod
def execute(self):
 pass
class AddOperation(AbstractOperation):
 def execute(self):
 print("Add=",self.operand_a + self.operand_b)
class SubtractOperation(AbstractOperation):
 def execute(self):
 print("Sub=",self.operand_a - self.operand_b)
class MultiplyOperation(AbstractOperation):
 def execute(self):
 print("Mul=",self.operand_a*self.operand_b)
class DivideOperation(AbstractOperation):
 def execute(self):
 print("Div=",self.operand_a/self.operand_b)
operation = AddOperation(1, 2)
operation.execute()
operation = SubtractOperation(8, 2)
operation.execute()
operation = MultiplyOperation(8, 2)
operation.execute()
operation = DivideOperation(8, 2)
operation.execute()
```

Interface in Python:-

An interface acts as a template for designing classes. Interfaces also define methods the same as classes, but abstract methods, whereas class contains nonabstract methods. Abstract methods are those methods without implementation or which are without the body. So the

interface just defines the abstract method without implementation. The implementation of these abstract methods is defined by classes that implement an interface.

How to Create Interface in Python?

There are two ways in python to create and implement the interface,

1. Informal Interfaces
2. formal Interfaces

1.Informal Interfaces

python informal interface is also a class that defines methods that can be overridden but without force enforcement. An informal interface also called Protocols or Duck Typing. The duck typing is actually we execute a method on the object as we expected an object to have, instead of checking the type of an object. If its beaver is the same as we expected, then we will be fine and go farther, else if it does not, things might get wrong, and for safety, we use a try..except block or hasattr to handle the exceptions to check the object have the particular method or not.

An informal interface in python is termed as a protocol because it is informal and cannot be formally enforced. It is mostly defined by templates or demonstrates in the documentations. Consider some of the methods we usually used – __len__, __iter__, __contains__ and all, which are used to perform some operation or set of protocols.

Example:-

```
class Fruits :
    def __init__( self, ele) :
        self.__ele = ele
    def __contains__( self, ele) :
        return ele in self.__ele
    def __len__( self ):
        return len( self.__ele)
fruits_list = Fruits([ "Apple", "Banana", "Orange" ])
print(len(Fruits_list))
print("Apple" in Fruits_list)
print("Mango" in Fruits_list)
print("Orange" not in Fruits_list)
```

2. Formal Interfaces

A formal Interface is an interface which enforced formally. In some situations, the protocols or duck typing creates confusion, like consider the example we have two classes FourWheelVehicle and TwoWheelVehicle both have a method SpeedUp( ), so the object of both class can speedup, but both objects are not the same even if both classes implement the same

interface. So to resolve this confusion, we can use the formal interface. To create a formal interface, we need to use ABCs (Abstract Base Classes).
An ABC is simple as an interface or base classes define as an abstract class in nature and the abstract class contains some methods as abstract. Next, if any classes or objects implement or drive from these base classes, then these bases classes forced to implements all those methods. Note that the interface cannot be instantiated, which means that we cannot create the object of the interface. So we use a base class to create an object, and we can say that the object implements an interface. And we will use the type function to confirm that the object implements a particular interface or not.

Example:-

```
import abc
class Myinterface( abc.ABC ) :
    @abc.abstractclassmethod
    def disp():
        pass
class Myclass( Myinterface ) :
    pass
o1 = Myclass( )
```

In the above code, the Myclass class inherits abstract class Myinterface but not provided the disp() abstract method's implementation. The class Myclass also becomes an abstract class and hence cannot be instantiated.

Example:-

An example of python code for a derived class defines an abstract method.

Code:

```
import abc
class Myinterface( abc.ABC ):
    @abc.abstractclassmethod
    def disp( ):
        pass
#print(" Hello from Myclass ")
class Myclass(Myinterface):
    def disp( ):
        pass
o1=Myclass()
```

Example 2

Example of python code for the derived class which defines an abstract method with the proper definition –

Code:

```
import abc
```

```
class Myinterface( abc.ABC ):
    @abc.abstractclassmethod
    def disp( ):
        pass
#print(" Hello from Myclass ")
class Myclass(Myinterface):
    def disp(s):
        print(" Hello from Myclass ")
o1=Myclass()
o1.disp()
```

Example 3

Example of python code to understand object types –

Code:

```
import abc
class FourWheelVehicle (abc.ABC):
    @abc.abstractmethod
    def SpeedUp( self ):
        pass
class Car(FourWheelVehicle) :
    def SpeedUp(self):
        print(" Running! ")
s = Car()
print( isinstance(s, FourWheelVehicle))
```

Example 4

Example of python code abstract class implemented by multiple derive classes –

Code:

```
import abc
class FourWheelVehicle (abc.ABC):
    @abc.abstractmethod
    def SpeedUp( self ):
        pass
class Car(FourWheelVehicle) :
    def SpeedUp(self):
        print(" Running! ")
class TwoWheelVehicle (abc.ABC) :
    @abc.abstractmethod
```

```
    def SpeedUp(self):
        pass
class Bike(TwoWheelVehicle) :
    def SpeedUp(self) :
        print(" Running!.. ")
a = Bike ()
s = Car()
print( isinstance(s, FourWheelVehicle))
print( isinstance(s, TwoWheelVehicle))
print( isinstance(a, FourWheelVehicle))
print( isinstance(a, TwoWheelVehicle))
```