PYTHON OOP

Object Oriented Programing

Object-oriented programming (OOP) is to design program using class and object.

Class: It is collection of objects. It is a logical entity that has some specific attributes and methods. In python class can be created using class keyword.

```
class Test:
    x=10
```

Object: It can be defined as an instance of a class. It can contain both the data and the function of class. In python object can be created by using class name.

```
t=Test()
print(t.x)
Here t is object of class Test, variable of the class can be accessed by the object.
```

OOP Properties

OOP is based on four basic concepts:

1. **Inheritance** : The ability to create a new class from an existing class is called Inheritance. Using inheritance, we can create a Child class from a Parent class such that it inherits the properties and methods of the parent class and can have its own additional properties and methods.
2. **Abstraction** : It refers to, providing only essential information to the outside world and hiding their background details.
3. **Encapsulation** : It is restriction of access to methods and variables. This prevents data from direct modification which is called encapsulation. Encapsulation is a process of binding data members and member functions into a single unit.
4. **Polymorphism** : Polymorphism is an ability to use a common interface for multiple forms.

__init__() and self

Functions that begin with double underscore __ are called special functions as they have special meaning.

__init__() is special function which is executed when object of that class in initiated. It can be related to constructors which are used to initializing the object values. This function is called automatically every time when a new object is created.

Self is a parameter which refers to the current object of the class. It is used to access the variables of the class. It has to be the first parameter of any function in the class.

Constructor:

```
class Car:
    def __init__(self,wheels,millage):      #Takes parameters; called parameterized constructor
        self.whells=wheels
        self.millage=millage
        print("Id of self:",id(self)) #1986115670872

c1=Car(4,60)
print(c1.wheels)      #4
print(c1.mileage)     #60
print("Id of object:",id(c1))    #1986115670872
```

-> It is clearly seen that self and object is referring to same object.

Object Methods:

```
class Student:
    def __init__(self,sid,name):
        self.sid=sid
        self.name=name
    def show(self):
        print("Id:",self.sid)
        print("Name:",self.name)
s1=Student(101,'Vivek')
s1.show()
```

Default Constructor

It is a simple constructor which doesn’t accept any arguments, the constructor which accept arguments are called parameterized constructor (In previous slide).

```
class Student:
    def __init__(self):
        self.name='Anand'
    def show(self):
        print("Name:",self.name)

s1=Student()
s1.show()
```

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.

```
class Student:
    def __init__(self):
        print("Student object created...")  #When object is created
    def __del__(self):
        print("Student object deleted...")  #When object deleted

s1=Student()     #Call __init__()
del s1           #Call __del__()
```

Inheritance

It is to define a class that inherits all the methods and properties from another class.The new class is called **derived** (or child) class and the one from which it inherits is called the **base** (or parent) class.
It provides **reusability** of a code. We don't have to write the same code again and again.

```
Single Inheritance:
When a child class inherits from only one parent class, it is called single inheritance

class Employee:                       #Parent Class
    def __init__(self,eid,ename):
        self.eid=eid
        self.ename=ename
    def show(self):
        print("Employee Name:",self.ename)

class Developer(Employee):            Child Class
    def display(self):
        print("Id of developer:",self.eid)

d1=Developer(101,'Ram')
d1.show()                    #Employee Name: Ram
d1.display()                 #Id of developer: 101
```

Multiple Inheritance:

When a child class inherits from multiple parent classes, it is called multiple inheritance.

```
class Employee:             #Base Class 1
    eid=101
class Trainer:              #Base Class 2
    tid=999
class Developer(Employee,Trainer):      #Child Class
    did=1001
    def show(self):
        print("eid",self.eid)
        print("eid",self.tid)
        print("eid",self.did)
d1=Developer()
d1.show()
```

Op:

```
eid: 101
tid: 999
did: 1001
```

Multilevel Inheritance:

Parent-child-grandchild level relationship is called multi level relationship.

```
#Parent Class
class Human:
    def beingHuman(self):
        print("I am a human!")
#Child Class
class Emp(Human):
    def beingEmp(self):
        print("I am an Employee!")
#Grandchild Class
class Developer(Emp):
    def beingDev(self):
        print("I am a developer!")

d1=Developer()
d1.beingHuman()  #I am a human!
d1.beingEmp()    #I am an Employee!
d1.beingDev()    #I am a developer!
```

Inherit Private Members

In Python, private attributes are denoted using underscore as the prefix i.e single _ or double __. By using private attributes in parent class inheritance can be prevented in child class.

```
class Father:
    def __init__(self):
        self.car=50000
        self.__home=70000           #Declaring private attribute
    def myproperty(self):
        print(self.car)             #50000
        print(self.__home)          #70000
class Son(Father):
    def inherited_property(self):
        print(self.car)             #50000
        print(self.__home)          #Error

s1=Son()
s1.inherited_property()     #Error "Son has no attribute home", only shows the car value
f1=Father()
f1.myproperty()             #Will show both car and home value
```

Encapsulation

It wraps data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly.

```
class Bike:
    def __init__(self):
        self.__price=80000
    def showPrice(self):
        print("Price:",self.__price)
    def setPrice(self,new_price):
        self.__price=new_price

b1=Bike()
b1.__price=75000      #Modify the price directly
b1.showPrice()        #Price: 80000
b1.setPrice(90000)    #Modify price using member function
b1.showPrice()        #Price: 90000
```

Here, price can't be modified without calling the member function as price is private attribute.

Polymorphism

The use of same function name (but different signatures) being uses for different task is called polymorphism. This can be called function overloading.

Simple Example to demonstrate Polymorphism:

```
def sum(x,y,z=0):
    return x+y+z
print(sum(5,6))       #Calling same function with 2 arguments
print(sum(7,8,5))     #Calling same function with 3 arguments
```

Polymorphism with class Methods:

```
class Odisha:
    def capital(self):
        print("Capital of Odisha : Bhubaneswar")
class Karntaka:
    def capital(self):
        print("Capital of Karnataka : Bangalore")

od=Odisha()
od.capital()
ka=Karntaka()
ka.capital()
```

Method Overriding:

When we use same method name with inheritance (polymorphism with inheritance), in such case we re-implement the polymorphic function. This process of re-implementing a method in the child class is known as **Method Overriding**.

```
class Country:
    def say(self):
        print("India is a country")
class India(Country):
    def say(self):
        print("India has many states")

ind=India()
ind.say()
```

Here the say() method will override by the child class. So it is called overridden function.

Operator Overloading

In python the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings. This feature in OOP that allows the same operator to have different meaning according to the context is called operator overloading. It can be achieved by using special functions.

```
class Sum:
    def __init__(self,a):
        self.a=a
    def __add__(self, other):
        return self.a+other.a

s1=Sum(5)
s2=Sum(6)
print(s1+s2)   #11
```

Advantages of OOP

- Object-Oriented Programming makes the program easy to understand as well as efficient.
- As class is sharable, the code can be reused.
- Data is safe and secure with data abstraction.
- Polymorphism allows the same interface for different objects, so programmers can write efficient code.

Hands-On 6

1. Create an “Animal” class and specify its attributes, then inherit the class “Tiger” and specify its special Attributes.
2. Create Three class “Space”, “Galaxy” and “Earth” and print variables of Space and Galaxy class in Earth class.
3. Calculate area of Square, Rectangle and triangle by using same function “area()” .
4. Overload * operator using Operator Overloading.

INVOKING PYTHON SCRIPT

__name__ and __main__

In python special variables are denoted by appending __ (Double underscore).

__name__:
When Python interpreter reads a source file it sets the this special variable. It is also a global variable.

__main__:
If the python interpreter is running that module (the source file) as the main program, it sets the special __name__ variable to have a value “__main__”. If this file is being imported from another module, __name__ will be set to the module’s name.

```
if __name__=='__main__':
    print("I am in main...")
```

Invoking The main method:

The special variables are used to invoke the main method of a program.

```python
def main():
    print("Main program start...")

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

Advantages:

- Every Python module has it’s __name__ defined and if this is ‘__main__’, it implies that the module is being run standalone by the user
- If we import this script as a module in another script, the __name__ is set to the name of the script/module.
- Python files can act as either reusable modules, or as standalone programs.
- It protects users from accidentally invoking the script when they didn't intend to.