NETWORKZ SYSTEMS
AN ISO 9001:2015 CERTIFIED COMPANY
CHINNAKADA , KOLLAM

OOPS IN PYTHON

OOPs in Python

Python is a multi-paradigm programming language.Meaning, it supports different programming approach.

One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

In Python, the concept of OOP follows some basic principles:

OOPs in Python

Object

The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Parrot is an object,

name, age, color are state/attributes

singing, dancing are behavior/methods

Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods.For example: if you have an employee class then it should contain an attribute and method, i.e. an email id, name, age,salary, etc.

Method

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods

Inheritance

A process of using details from an existing class to a new class without modifying existing class.

Inheritance is the most important aspect of object-oriented programming which simulates the real world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class. It provides re-usability of the code.

Polymorphism

A concept of using common operation in different ways for different data input.

By polymorphism, we understand that one task can be performed in different ways. For example You have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in the sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

Encapsulation

Wrapping up of data and functions in a single unit.

Encapsulation is also an important aspect 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

Data Abstraction

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

Create a Class

To create a class, use the keyword class

```
class ClassName:
        #statement_suite
```

Example

```
class Fruits:
        name= 'Apple'
```

```
class Cat:
        age=5
```

Create an Object

Syntax: object-name= class-name(arguments)

```
class MyClass:
        x=5


p1 = MyClass()
print(p1.x)//5
```

Python Constructor

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.

- In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

- Constructors can be of two types.
  - Parameterized Constructor
  - Non-parnameterized Constructor
- Constructor definition is executed when we create the object of this class.

- It also verify that there are enough resources for the object to perform any start-up task

The _ _init_ _() Function

All classes have a function called __init__(), which is always executed when the class is being initiated.

It accepts the self-keyword as a first argument which allows accessing the attributes or method of the class.

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

Creating Class and Object in Python

```
class Employee:
        id = 10
        name = "ayush"
        def display (self):
                print(self.id,self.name)

emp = Employee()
emp.display()
```

Methods

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

```
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)
print(blu.sing("Happy")) # call our instance methods
print(blu.dance())
```

Output

Blu sings 'Happy'
Blu is now dancing

In the above program, we define two methods i.e sing() and dance(). These are called instance method because they are called on an instance object i.e blu.

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.

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

p1 = Person("John", 36)
p1.myfunc()
```

Inheritance

Inheritance is a way of creating new class for using details of 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).

Syntax:

class derive-class(<base class1>,<base class 2>,.....<base class n>):
<class-suite>

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

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from swim()method. Again, the child class modified the behavior of parent class. We can see this from whoisThis() method. Furthermore, we extend the functions of parent class, by creating a new run() method. Additionally, we use super() function before __init__() method. This is because we want to pull the content of __init__() method from the parent class into the child class.

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix

i.e single “ _ “ or double “ _ _“.

Public Attributes

```
class Employee:
        def __init__(self, name, sal):
            self.name=name
            self.salary=sal

e1=Employee("Kiran",10000)
print(e1.salary)//10000
e1.salary=20000
print(e1.salary)//20000
```

Private Attributes

double underscore __ prefixed to a variable makes it private. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:

```
class employee:
    def __init__(self, name, sal):
        self ._  _ name=name # private attribute
        self._ _salary=sal #private attribute
e1=employee("Bill",10000)
print(e1.__salary)
AttributeError: 'employee' object has no attribute '__salary'
```

Private Attributes

```
class Computer:
        def __init__(self):
                self.__maxprice = 900
        def sell(self):
                print("SellingPrice:{}".format(self.__maxprice))
        def setMaxPrice(self, price):
                self.__maxprice = price

c = Computer()
c.sell()
c.__maxprice = 1000  # change the price
c.sell()
c.setMaxPrice(1000)# using setter function
c.sell()
```

Private Attributes

Output

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

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 parameter.

Protected Attributes

Python's convention to make an instance variable protected is to add a prefix _ (single underscore) to it. This effectively prevents it to be accessed, unless it is from within a sub-class.

```
class employee:
    def __init__(self, name, sal):
            self._name=name # protected attribute
            self._salary=sal # protected attribute

e1=employee("Swati", 10000)
print(e1._salary)        o/p : 10000
e1._salary=20000
Print(e1._salary)       o/p : 20000
```

Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types). Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism

Polymorphism

```
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()
```

Polymorphism

```
#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

Output

```
Parrot can fly
Penguin can't fly
```

Polymorphism

In the above program, we defined two classes Parrot and Penguin. Each of them have common method fly() method. However, their functions are different. To allow polymorphism, we created common interface i.e flying_test() function that can take any object. Then, we passed the objects blu and peggy in the flying_test() function, it ran effectively.

THANK YOU

NETWORKZ SYSTEMS

AN ISO 9001:2015 CERTIFIED COMPANY

CHINNAKADA , KOLLAM