Python Classes and Objects

A class is a collection of objects or you can say it is a blueprint of objects defining the common attributes and behavior.

Example:

Class class1(): // class 1 is the name of the class

Note: Python is not case-sensitive.

Objects:

Objects are an instance of a class. It is an entity that has state and behavior. In a nutshell, it is an instance of a class that can access the data.

Syntax: obj = class1()

Here obj is the "object " of class1.

Creating an Object and Class in python:

Example:

```
class employee():
        def  __init__(self,name,age,id,salary):     //creating  a
function
        self.name = name // self is an instance of a class
        self.age = age
        self.salary = salary
        self.id = id

emp1 = employee("harshit",22,1000,1234) //creating objects
emp2 = employee("arjun",23,2000,2234)
print(emp1.__dict__)//Prints dictionary
```

Explanation: 'emp1' and 'emp2' are the objects that are instantiated against the class 'employee'.Here, the word (__dict__) is a "dictionary" which prints all the values of object 'emp1' against the given parameter (name, age, salary).(__init__) acts like a constructor that is invoked whenever an object is created.

Object-Oriented Programming methodologies:

Object-Oriented Programming methodologies deal with the following concepts

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

Inheritance:

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Single Inheritance:

Single level inheritance enables a derived class to inherit characteristics from a single parent class.

Example:

```
class employee1()://This is a parent class
def __init__(self, name, age, salary):
self.name = name
self.age = age
self.salary = salary

class childemployee(employee1)://This is a child class
def __init__(self, name, age, salary,id):
self.name = name
self.age = age
self.salary = salary
self.id = id
emp1 = employee1('harshit',22,1000)

print(emp1.age)
```

Output: 22

Explanation:

- I am taking the parent class and created a constructor (__init__), class itself is initializing the attributes with parameters(‘name’, ‘age’ and ‘salary’).
- Created a child class ‘childemployee’ which is inheriting the properties from a parent class and finally instantiated objects ’emp1′ and ’emp2′ against the parameters.

Multilevel Inheritance:

Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

Example:

```
class employee()://Super class
def __init__(self,name,age,salary):
self.name = name
self.age = age
self.salary = salary
class childemployee1(employee)://First child class
def __init__(self,name,age,salary):
self.name = name
self.age = age
self.salary = salary

class childemployee2(childemployee1)://Second child class
def __init__(self, name, age, salary):
self.name = name
self.age = age
self.salary = salary
emp1 = employee('harshit',22,1000)
emp2 = childemployee1('arjun',23,2000)

print(emp1.age)
print(emp2.age)
```

Output: 22,23

Explanation:

- It is clearly explained in the code written above, Here I have defined the superclass as employee and child class as childemployee1. Now, childemployee1 acts as a parent for childemployee2.

- I have instantiated two objects ’emp1′ and ’emp2′ where I am passing the parameters “name”, “age”, “salary” for emp1 from superclass “employee” and “name”, “age, “salary” and “id” from the parent class “childemployee1”

Hierarchical Inheritance: Hierarchical level inheritance enables more than one derived class to inherit properties from a parent class.

Example:

```
class employee():
def __init__(self, name, age, salary):  //Hierarchical
Inheritance
self.name = name
self.age = age
self.salary = salary

class childemployee1(employee):
def __init__(self,name,age,salary):
self.name = name
self.age = age
self.salary = salary

class childemployee2(employee):
def __init__(self, name, age, salary):
self.name = name
self.age = age
self.salary = salary
emp1 = employee('harshit',22,1000)
emp2 = employee('arjun',23,2000)

print(emp1.age)
print(emp2.age)
```

Output: 22,23

Explanation:

- In the above example, you can clearly see there are two child class “childemployee1” and “childemployee2”. They are inheriting functionalities from a common parent class that is “employee”.
- Objects ’emp1′ and ’emp2′ are instantiated against the parameters ‘name’, ‘age’, ‘salary’.

Multiple Inheritance: Multiple level inheritance enables one derived class to inherit properties from more than one base class.

Example:

```
class employee1()://Parent class
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

class employee2()://Parent class
    def __init__(self,name,age,salary,id):
     self.name = name
     self.age = age
     self.salary = salary
     self.id = id

class childemployee(employee1,employee2):
    def __init__(self, name, age, salary,id):
     self.name = name
     self.age = age
     self.salary = salary
     self.id = id
emp1 = employee1('harshit',22,1000)
emp2 = employee2('arjun',23,2000,1234)

print(emp1.age)
print(emp2.id)
```

Output: 22,1234

Explanation: In the above example, I have taken two parent class “employee1” and “employee2”.And a child class “child employee”, which is inheriting both parent class by instantiating the objects ’emp1′ and ’emp2′ against the parameters of parent classes.

Polymorphism:

You all must have used GPS for navigating the route, Isn’t it amazing how many different routes you come across for the same destination depending on the traffic, from a programming point of view this is called ‘polymorphism’. It is one such OOP methodology where one task can be performed in several different ways. To put it in simple words, *it is a property of an object which allows it to take multiple forms*.

- *Compile-time Polymorphism*
- *Run-time Polymorphism*

Compile-time Polymorphism: A compile-time polymorphism also called as static polymorphism which gets resolved during the compilation time of the program. One common example is “method overloading”.

Example:

```
class employee1():
def name(self):
print("Harshit is his name")
def salary(self):
print("3000 is his salary")

def age(self):
print("22 is his age")

class employee2():
def name(self):
print("Rahul is his name")

def salary(self):
print("4000 is his salary")

def age(self):
print("23 is his age")

def func(obj)://Method Overloading
obj.name()
obj.salary()
obj.age()

obj_emp1 = employee1()
obj_emp2 = employee2()

func(obj_emp1)
func(obj_emp2)
```

Output:

```
Harshit is his name
3000 is his salary
22 is his age
Rahul is his name
4000 is his salary
23 is his age
```

Run-time Polymorphism:

A run-time Polymorphism is also, called as dynamic polymorphism where it gets resolved into the run time. One common example of Run-time polymorphism is “method overriding”.

Example:

```
class employee():
   def __init__(self,name,age,id,salary):
       self.name = name
       self.age = age
       self.salary = salary
       self.id = id
def earn(self):
        pass

class childemployee1(employee):

   def earn(self)://Run-time polymorphism
      print("no money")

class childemployee2(employee):

   def earn(self):
       print("has money")

c = childemployee1
c.earn(employee)
d = childemployee2
d.earn(employee)
```

Output: no money, has money

Encapsulation:

In a raw form, encapsulation basically means binding up of data in a single class. Python does not have any private keyword, unlike Java. A class shouldn’t be directly accessed but be prefixed in an underscore.

Example:

```
class employee(object):
def __init__(self):
self.name = 1234
self._age = 1234
self.__salary = 1234

object1 = employee()
print(object1.name)
print(object1._age)
print(object1.__salary)
```

Output:

```
1234
Traceback (most recent call last):
1234
File "C:/Users/Harshit_Kant/PycharmProjects/test1/venv/encapsu.py", line
10, in
print(object1.__salary)
AttributeError: 'employee' object has no attribute '__salary'
```

Explanation: You will get this question what is the underscore and error? Well, python class treats the private variables as(__salary) which can not be accessed directly.

So, I have made use of the setter method which provides indirect access to them in my next example.

Example:

```
class employee():
def __init__(self):
self.__maxearn = 1000000
def earn(self):
print("earning is:{}".format(self.__maxearn))

def setmaxearn(self,earn)://setter method used for accesing
private class
self.__maxearn = earn

emp1 = employee()
emp1.earn()

emp1.__maxearn = 10000
emp1.earn()

emp1.setmaxearn(10000)
emp1.earn()
```

Output:

earning is:1000000,earning is:1000000,earning is:10000

Abstraction:

Suppose you booked a movie ticket from bookmyshow using net banking or any other process. You don't know the procedure of how the pin is generated or how the verification is done. This is called 'abstraction' from the programming aspect, it basically means you only show the implementation details of a particular process and hide the details from the user. It is used to simplify complex problems by modeling classes appropriate to the problem.

An abstract class cannot be instantiated which simply means you cannot create objects for this type of class. It can only be used for inheriting the functionalities.

Example:

```
from abc import ABC,abstractmethod
class employee(ABC):
def emp_id(self,id,name,age,salary):    //Abstraction
pass

class childemployee1(employee):
def emp_id(self,id):
print("emp_id is 12345")

emp1 = childemployee1()
emp1.emp_id(id)
```

Output: emp_id is 12345

The pass Statement

`class` definitions cannot be empty, but if you for some reason have a `class` definition with no content, put in the `pass` statement to avoid getting an error.

Example

```
class Person:

  Pass
```