Unit V
Object Oriented Programming in Python

Introduction

- Python is an Object- Oriented Programming Language that follows the concept of Object Oriented Programming Paradigm.
- It deals with declaring Python classes and objects which lays the foundation of OOP concepts.
- Python Programming offers OOP style programming and provides an easy way to develop programs.
- Python uses the OOP concepts that makes python more powerful to help design a program that design a program that represents real world entities
- Python also supports OOP concept such as Inheritance, Method Overriding, data abstraction, etc.

Important terms in OOP

- **Class:** Classes are defined by the user. The class provides basic structure for an object. It consist of data methods and method members that are used by the instances of the class.
- **Object:** A unique instance of a data structure that is defined by its class. An object comprises both data members and methods.
- **Data Members:** A variable defined in either a class or an object; it holds the data associated with the class or object.
- **Instance Variable:** A variable that is defined in a method; its scope is only within the object that defines it.
- **Class variable:** A variable that is defined in the class and can be used by all the instances of that class.

- **Function Overloading:** A function defined more than one time with different behaviour is known as function overloading.
- **Encapsulation:** It is the process of binding together the methods and data variables as a single entity.
- **Inheritance:** The transfer of characteristics of a class to another class that are derived from it.
- **Polymorphism:** It allows one interface to be used for a set of actions i.e, one name may refer to many functionalities.

Classes

- Python is an object oriented programming language. Almost everything in python is an object, with its properties and methods.
- Object is simple a collection of data(variables) and methods (functions) that act on those data.
- A class is like an object constructor or a blueprint for creating objects.
- A class defines the properties and behaviour that is shared by all the objects.

Creating Classes

- A class is a block of statements that combine data and operations which are performed on the data, into a group as a single unit and acts as a blueprint for the creation of objects.
- To create a class, use the keyword ‘class’.

Syntax:

```
Class class_name:
        #list of python variables
        #python class constructor
        #python class method
```

Example:

```
class car:
    pass
```

Here, pass is the statement used to indicate that the class is empty

- In a class we can define variables, functions, etc. while writing any function in class we have to pass at least one argument that is called as Self Parameter
- The **self parameter** is a reference to the class itself and it is used to access variables that belongs to the class.

Example:

```
class student:
    def display(self):
        print("Hello Python")
```

Objects and Creating Objects

- An object is an instance of a class that has some attributes and behaviour.
- Objects can be used to access the attributes of the class.

Syntax:

```
obj_name = class_name()
```

Example:

```
s1=student()
s1.display()
```

Example

```
class student:
        def display (self):
                print(“Hello Python”)
s1= student()
s1.display()
```

Output:

Hello Python

Constructor

- A constructor is a special method i.e., used to initialize the instance of a class

Creating constructor in Class:

- A constructor is a special type of method which is used to initialize the instance variable of the class.
- Constructors are generally used for instantiating an object. The task of the constructor is to initialize the data members of the class when an object of the class is created.
- Python class constructor is the first piece of code to be executed when we create a new object of a class.
- In python the __init__() method is called the constructor and is always called when an object is created.

- The constructor can be used to put values in the member variables.

Syntax:

```
def __init__(self):
    #body of the constructor
```

- __init__ is a special method in python which is the constructor method for a class.

Types of Constructors in Python

- Parameterized Constructor
- Default Constructor

1. Default Constructor

- When you do not write the constructor in the class created, Python itself creates a constructor during the compilation of the program.
- It generates an empty constructor that has no code in it.

Example:

```
class student:
        def __init__(self):
                print("default constructor")
        def show(self,name):
                print("hello",name)
s1=student()
s1.show("ABC")
```

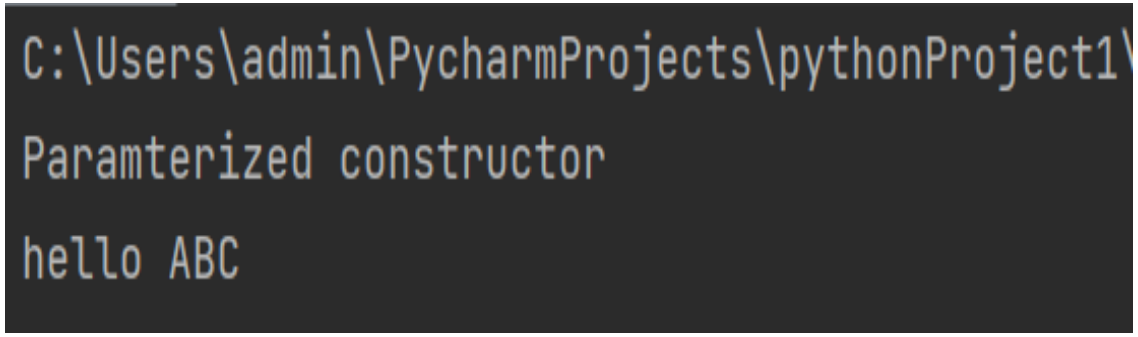
```
C:\Users\admin\PycharmProjects\pythonProject1
default constructor
hello ABC
```

2. Parameterized Constructor

- When the constructor accepts arguments along with self, it is known as parameterized constructor.
- These arguments can be used inside the class to assign the values to the data members.

Example:

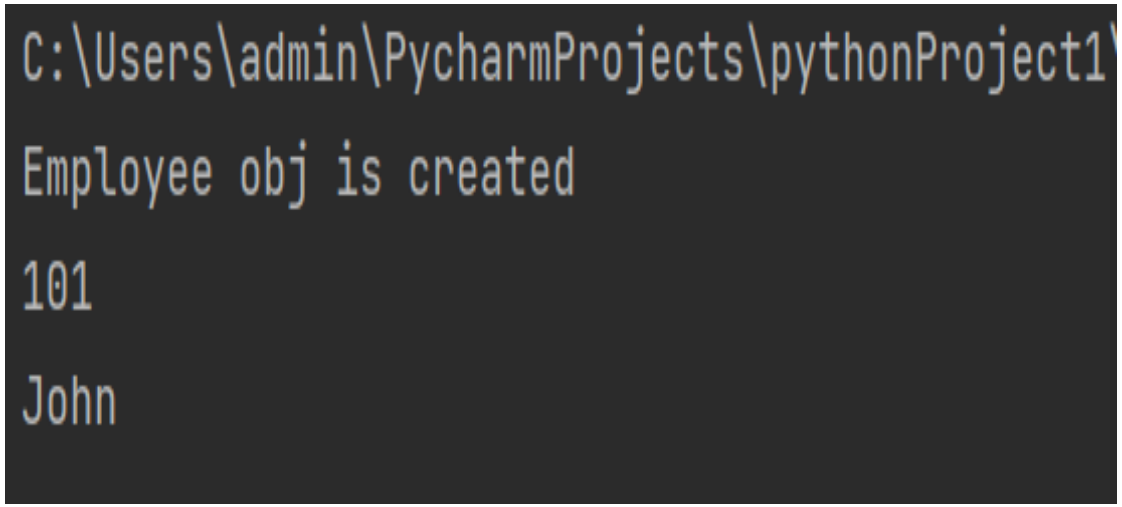
```
class student:
        def __init__(self,name):
                print("Paramterized constructor")
                self.name=name
        def show(self):
                print("hello",self.name)
s1=student("ABC")
s1.show()
```



Example

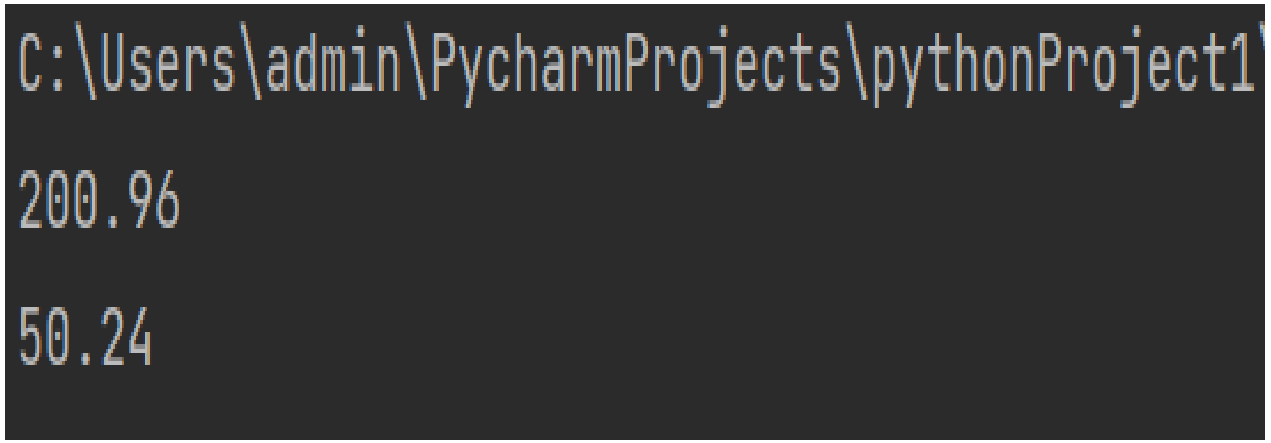
1. For creating constructor

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name
emp1 = Employee("John", 101)
print("Employee obj is created")
print(emp1.id)
print(emp1.name)
```



2. Create a class circle and initialize it with radius. Make two methods inside the class

```
class Circle():
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius ** 2 * 3.14
    def perimeter(self):
        return 2 * self.radius * 3.14
NewCircle = Circle(8)
print(NewCircle.area())
print(NewCircle.perimeter())
```



Destructor

- Destructors are called when an object gets destroyed.
- In Python, destructors are not needed as much as 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.
- It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax:

```
def __del__(self):
    # body of destructor
```

Example:

```
class Employee:
        def __init__(self):
                print('Employee created.')
        def __del__(self):
                print('Destructor called, Employee deleted.')
obj = Employee()
del obj
```

```
C:\Users\admin\PycharmProjects\pythonProject1
Employee created.
Destructor called, Employee deleted.
```

Method Overloading

- Two or more methods have the same name but different numbers of parameters or different types of parameters, or both.
- These methods are called overloaded methods and this is called method **overloading**.
- Like other languages do, python does not support method overloading by default. But there are different ways to achieve method overloading in Python.
- The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.

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

```
C:\Users\admin\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Users\admin
Traceback (most recent call last):
  File "C:\Users\admin\PycharmProjects\pythonProject1\main.py", line 9, in <module>
    product(4, 5)
TypeError: product() missing 1 required positional argument: 'c'
```

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

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

```
C:\Users\admin\PycharmProjects\pythonProject1\venv\Scripts\python.exe
100
```

We can achieve method overloading in python by user defined function using "**None**" keyword as default parameter.

```
def add(a=None, b=None):
    if a != None and b == None:
        print(a)
    # else will be executed if both are available and returns addition of two
    else:
        print(a + b)
add(2, 3)
add(2)
```

```
C:\Users\admin\PycharmProjects\pythonProject1\venv\Scripts\python.exe
5
2
```

With a method to perform different operation using method overloading

```
class operation:
    def add (self,a,b):
        return a+b
op1=operation()
print("Addition=", op1.add(10,20))
print("Addition=", op1.add(10.12,20.13))
print("Addition=", op1.add("hello","python"))
```

```
C:\Users\admin\PycharmProjects\pythonProject1\venv\Scripts\python.exe
Addition= 30
Addition= 30.25
Addition= hellopython
```

Inheritance

- In inheritance object of one class procure the properties of object of another class.
- Inheritance provides code reusability which means that some of the new features can be added to the code while using the existing code. This mechanism of designing or constructing classes from other classes is called inheritance
- The new class is called as derived class or child class and the class from which this derived class have been inherited is the base class or parent class.
- In Inheritance the child class acquire the properties and can access all the data members and functions defined in the parents class.

Single Inheritance

Syntax:

```
class A:
        # properties of class A
class B (A):
        # properties of class B
```

Inheritance without using constructor

```
class vehicle:
    name="abc"
    def display(self):
        print("name=",self.name)
class category (vehicle):
    price=2000
    def dis_price(self):
        print("Price=", self.price)
car1=category()
car1.display()
car1.dis_price()
```

```
C:\Users\admin\PycharmProjects\pythonProject1\

name= abc

Price= 2000
```

Inheritance using constructor

```
class vehicle:
    def __init__(self,name):
        self.name=name
    def display(self):
        print("name=",self.name)
class category (vehicle):
    def __init__(self,name,price):
        vehicle.__init__(self,name)   #passing data to base class constrcutor
        self.price=price
    def dis_price(self):
        print("price=",self.price)
car1=category("abc",30000)
car1.display()
car1.dis_price()
car2=category("xyz",30000)
car2.display()
car2.dis_price()
```

```
C:\Users\admin\PycharmProjects\pythonProject1\
name= abc
price= 30000
name= xyz
price= 30000
```

Multilevel Inheritance

- Multilevel inheritance is achieved when a derived class inherits another derived class.
- There is no limit on the number of levels up to which the multi-level inheritance is achieved in python.

Syntax:

```
class A:
        #properties of A
class B(A):
        #properties of B
class C(B):
        #properties of C
```

Example

```
class grandfather:
    def display(self):
        print("Grand Father")
class father(grandfather):
    def display2(self):
        print("Father")
class son (father):
    def display3(self):
        print("son")
s1=son()
s1.display()
s1.display2()
s1.display3()
```

```
C:\Users\admin\PycharmProjects\pythonProject1
Grand Father
Father
son
```

Multiple Inheritance

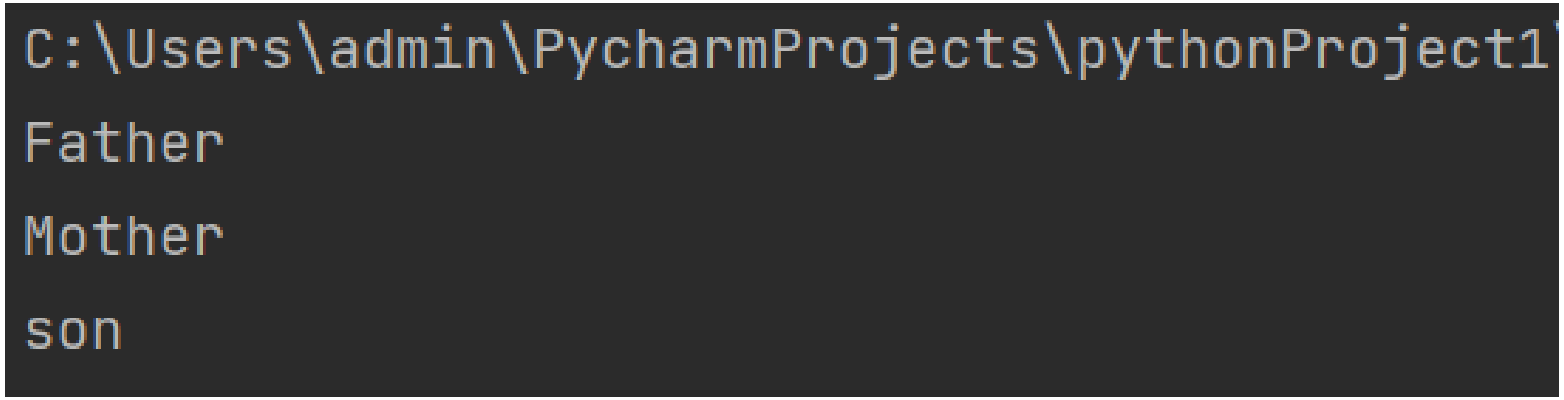
- Python provides flexibility to inherit multiple base classes in child class.
- Multiple inheritance means that we are inheriting the property of multiple classes into one. In case we have two classes A and B and we can create a new class that will inherit the properties of both A and B.

Syntax:

```
class A:
        #variables of class A
        #functions of class A
class B:
        #variables of class B
class C (A,B)
        #variables of class C
```

Example

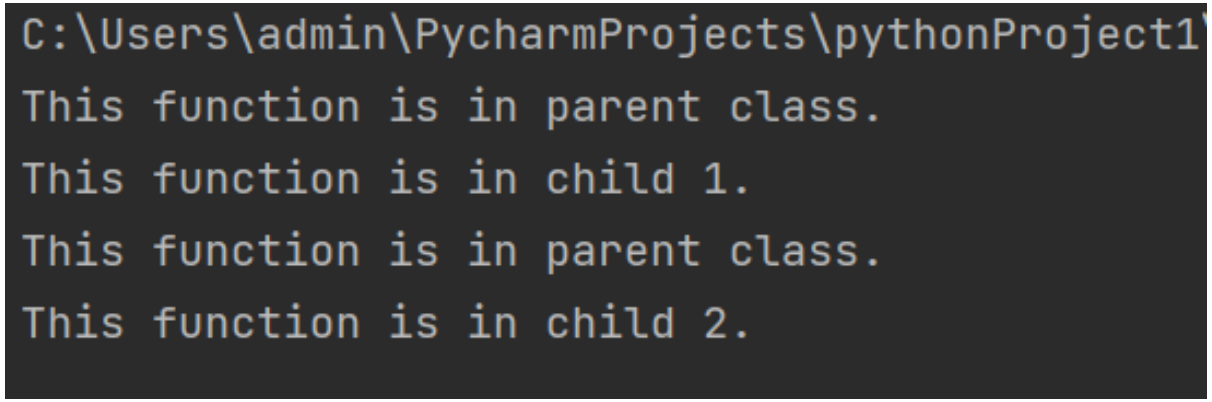
```
class father:
    def display(self):
        print("Father")
class Mother:
    def display2(self):
        print("Mother")
class son (father,Mother):
    def display3(self):
        print("son")
s1=son()
s1.display()
s1.display2()
s1.display3()
```



Hierarchical Inheritance

- When more than one derived classes are created from a single base class – it is called hierarchical inheritance.

```
class Parent:
        def func1(self):
        print("This function is in parent class.")
class Child1(Parent):
        def func2(self):
        print("This function is in child 1.")
class Child2(Parent):
        def func3(self):
        print("This function is in child 2.")
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

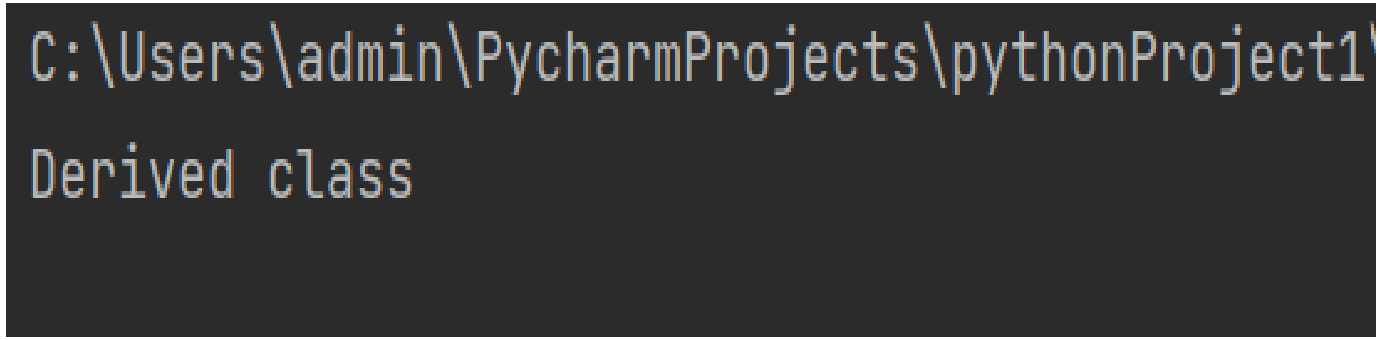


Method Overriding

- Overriding is the ability of a class to change the implementation of a method provide by one of its base class.
- To override a method in the base class, we must define a new method with same name and same parameters in the derived class.

Example:

```
class A:
        def display(self):
                print("base class")
class B(A):
        def display(self):
                print("Derived class")
obj=B()
obj.display()
```



Using super() method

- The super()method gives you access to methods in a superclass from the subclass that inherits from it.
- The super method also returns a temporary object of the superclass that then allows you to call that superclass's methods.

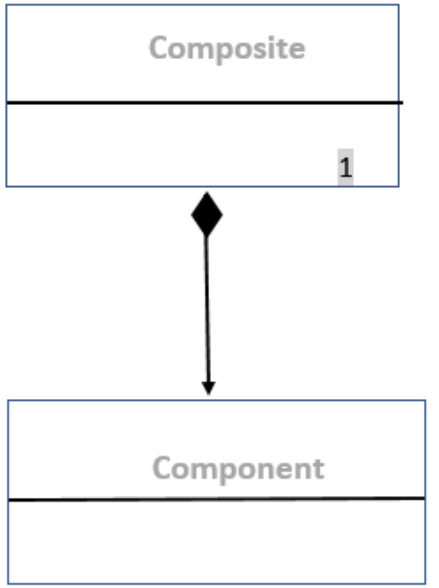
Example:

```
class A:
        def display(self):
                print("base class")
class B(A):
        def display(self):
                super().display()
                print("Derived class")
obj=B()
obj.display()
```

```
C:\Users\admin\PycharmProjects\pythonProject1
base class
Derived class
```

Composition classes

- In composition, we do not inherit from the base class but establish relationships between classes through the use of instance variables that are references to other object.
- Composition also reflects the relationship between parts called a “has-a” relationships.
- It enables complex types by combining objects of other types. This means that a class composite can contain an object of another class component.
- The composite side can express the cardinality of the relationship.
- The cardinality indicates the number or valid range of component instances the composite class will contain.



Syntax:

```
class GenericClass:
        #attributes and methods
class SpecificClass
        #instance_variable of Generic class
        #use this instance somewhere in the class
        some_method(instance variable of generic class)
```

Example:

```
class Component:
    def __init__(self):
        print('Component class object created...')
    def m1(self):
        print('Component class m1() method executed...')
class Composite:
    def __init__(self):
            self.obj1 = Component()
              print('Composite class object also created...')
    def m2(self):
        print('Composite class m2() method executed...')
        self.obj1.m1()


obj2 = Composite()
obj2.m2()
```

```
C:\Users\admin\PycharmProjects\pythonProject1\
Component class object created...
Composite class object also created...
Composite class m2() method executed...
Component class m1() method executed...
```

Experiment No:14

1. Write a Python program to create a class to print an integer and a character with two methods having the same name but different sequence of the integer and the character parameters. For example, if the parameters of the first method are of the form (int n, char c), then that of the second method will be of the form (char c, int n)

Solution:

```
def display(n, c):
    if n != None and c != None:
        print('Integer Value Using 1st Method =', n)
        print('Character Value Using 1st Method =', c)
def display(c, n):
    if c != None or n != None:
        print('Character Value Using 2nd Method =', c)
        print('Integer Value Using 2nd Method =', n)
n = int(input("Enter any integer value:"))
c = input("Enter any character value:")
 display(n, c)
display(c, n)
```

```
C:\Users\admin\PycharmProjects\pythonProject1
Enter any integer value:10
Enter any character value:a
Character Value Using 2nd Method = 10
Integer Value Using 2nd Method = a
Character Value Using 2nd Method = a
Integer Value Using 2nd Method = 10
```

- Write a Python program to create a class to print the area of a square and a rectangle. The class has two methods with the same name but different number of parameters. The method for printing area of rectangle has two parameters which are length and breadth respectively while the other method for printing area of square has one parameter which is side of square

Solution:

```
class Compute:
    def area(self,x = None, y = None):
        if x != None and y != None:
            return x * y
        elif x != None:
            return x * x
        else:
            return 0
l=int(input("Enter length of Rectangle:"))
b=int(input("Enter breadth of Rectangle:"))
s=int(input("Enter side of Rectangle:"))
obj = Compute()
print("Area of a rectangle:", obj.area(l, b))
print("Area of a square:", obj.area(s))
```

```
Enter length of Rectangle:10
Enter breadth of Rectangle:30
Enter side of Rectangle:4
Area of a rectangle: 300
Area of a square: 16
```

Write a Python program to create a class 'Degree' having a method 'getDegree' that prints "I got a degree". It has two subclasses namely 'Undergraduate' and 'Postgraduate' each having a method with the same name that prints "I am an Undergraduate" and "I am a Postgraduate" respectively. Call the method by creating an object of each of the three classes.

```
class degree():
    def getDegree(self):
        print("I got a degree")
class undergraduate(degree):
    def getDegree(self):
        print("I am a undergradute")
class postgraduate(degree):
    def getDegree(self):
        print("I am a post graduate")
a=degree()
b=undergraduate()
c=postgraduate()
a.getDegree()
b.getDegree()
c.getDegree()
```