Programming with Python & Java (CS 29008)

Lab 5

Object Oriented Programming in Python
Polymorphism

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY (KIIT)
Deemed to be University U/S 3 of UGC Act, 1956

School of Electronics Engineering
KIIT Deemed to be University
Bhubeneswar, Odisha

Contents

Lab 5: Polymorphism

1 Introduction

Polymorphism means ’Many Forms’.The various examples associated with polymorphism are

- Yourself is best example of polymorphism.In front of Your parents You will have one type of behaviour and with friends another type of behaviour.Same person but different behaviours at different places,which is nothing but polymorphism.
- + operator acts as concatenation and arithmetic addition.
- * operator acts as multiplication and repetition operator.
- The Same method with different implementations in Parent class and child classes (overriding).

Related to polymorphism the following 4 topics are important

1. Overloading
   (a) Operator Overloading
   (b) Method Overloading
   (c) Constructor Overloading
2. Overriding
   (a) Method overriding
   (b) constructor overriding

2 Operator Overloading:

We can use the same operator for multiple purposes, which is nothing but operator overloading. Python supports operator overloading.

1. + operator can be used for Arithmetic addition and String concatenation.

```
print(10+20)#30
print('John'+'Doe')
```

2. * operator can be used for multiplication and string repetition purposes.

```
print(10*20)#200
print('John'*3)#JohnJohnJohn
```

We can overload + operator to work with Book objects also. i.e Python supports Operator.

```
'''Program to use + operator for our class objects'''
class Book:
    def __init__(self,pages):
        self.pages=pages

>> b1=Book(100)
>> b2=Book(200)
>> print(b1+b2)
```

3 Method Overloading

If two methods having same name but different type of arguments then those methods are said to be overloaded methods.

```
m1(int a)
m1(double d)
```

But in Python Method overloading is not possible. If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method.

```
class Test:
    def m1(self):
        print('no-arg method')
    def m1(self,a):
        print('one-arg method')
    def m1(self,a,b):
        print('two-arg method')

>> t=Test()
>> t.m1()
>> t.m1(10)
>> t.m1(10,20)
```

In the above program python will consider only last method.

3.1 Handling overloaded method requirements in Python

Most of the times, if method with variable number of arguments required then we can handle with default arguments or with variable number of argument methods.

```
'''Program with Default Arguments'''
class Test:
    def sum(self,a=None,b=None,c=None):
        if a!=None and b!= None and c!= None:
            print('The Sum of 3 Numbers:',a+b+c)
        elif a!=None and b!= None:
            print('The Sum of 2 Numbers:',a+b)
        else:
            print('Please provide 2 or 3 arguments')

>> t=Test()
>> t.sum(10,20)
>> t.sum(10,20,30)
>> t.sum(10)  #Please provide 2 or 3 arguments
```

```
'''Program with Variable Number of Arguments'''
class Test:
    def sum(self,*a):
        total=0
        for x in a:
            total=total+x
        print('The Sum:',total)
>> t=Test()
>> t.sum(10,20)
>> t.sum(10,20,30)
>> t.sum(10)
>> t.sum()
```

4 Constructor Overloading

Constructor overloading is not possible in Python. If we define multiple constructors then the last constructor will be considered.

```
class Test:
    def __init__(self):
        print('No-Arg Constructor')

    def __init__(self,a):
        print('One-Arg constructor')

    def __init__(self,a,b):
        print('Two-Arg constructor')
>> t1=Test()
>> t1=Test(10,20)
```

In the above program only Two-Arg Constructor is available. But based on our requirement we can declare constructor with default arguments and variable number of arguments.

```
'''Constructor with Default Arguments'''
class Test:
    def __init__(self,a=None,b=None,c=None):
        print('Constructor with 0|1|2|3 number of arguments')

>> t1=Test()
>> t2=Test(10)
>> t3=Test(10,20)
>> t4=Test(10,20,30)
```

```
'''Constructor with Variable Number of Arguments'''
class Test:
    def __init__(self,*a):
        print('Constructor with variable number of arguments')

>> t1=Test()
>> t2=Test(10)
>> t3=Test(10,20)
>> t4=Test(10,20,30)
>> t5=Test(10,20,30,40,50,60)
```

5 Method Overriding

What ever members available in the parent class are by default available to the child class through inheritance. If the child class not satisfied with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding. Overriding concept applicable for both methods and constructors.

```
'''Program for Method overriding'''
class P:
    def property(self):
        print('Gold+Land+Cash+Power')
    def marry(self):
        print('Appalamma')

class C(P):
    def marry(self):
        print('Timmy')

>> c=C()
>> c.property()
>> c.marry()
```

From Overriding method of child class,we can call parent class method also by using super() method.

```
class P:
    def property(self):
        print('Gold+Land+Cash+Power')
    def marry(self):
        print('Appalamma')
class C(P):
    def marry(self):
        super().marry()
        print('Timmy')

>> c=C()
>> c.property()
>> c.marry()
```

```
'''Program for Constructor overriding'''
class P:
    def __init__(self):
        print('Parent Constructor')

class C(P):
    def __init__(self):
        print('Child Constructor')

>> c=C()
```

In the above example,if child class does not contain constructor then parent class constructor will be executed. From child class constructor we can call parent class constructor by using super() method.

```
'''Program to call Parent class constructor by using super()'''
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age

class Employee(Person):
    def __init__(self,name,age,eno,esal):
        super().__init__(name,age)
        self.eno=eno
        self.esal=esal

    def display(self):
        print('Employee Name:',self.name)
        print('Employee Age:',self.age)
        print('Employee Number:',self.eno)
        print('Employee Salary:',self.esal)
```

```
>> e1=Employee('John',48,872425,26000)
>> e1.display()
>> e2=Employee('Sunny',39,872426,36000)
>> e2.display()
```

6 Advanced Concepts

6.1 Public, Protected and Private Attributes

By default every attribute is public. We can access from anywhere either within the class or from outside of the class.
Eg: name = 'John'
Protected attributes can be accessed within the class anywhere but from outside of the class only in child classes. We can specify an attribute as protected by prefexing with _ symbol.
Syntax: _variablename = value
Eg: _name='John'
But is is just convention and in reality does not exists protected attributes.
Private attributes can be accessed only within the class.i.e from outside of the class we cannot access. We can declare a variable as private explicitly by prefexing with 2 underscore symbols.
syntax: __variablename=value Eg: __name='John'

```
class Test:
    x=10
    _y=20
    __z=30
    def m1(self):
        print(Test.x)
        print(Test._y)
        print(Test.__z)

>> t=Test()
>> t.m1()
>> print(Test.x)
>> print(Test._y)
>> print(Test.__z) #AttributeError: type #object 'Test' has no
                                  attribute '__z'
```

6.1.1 Accessing Private Variables from Outside of the Class

We cannot access private variables directly from outside of the class.But we can access indirectly as follows objectreference._classname__variablename

```
class Test:
    def __init__(self):
        self.__x=10
>> t=Test()
>> print(t._Test__x)#10
```

__str__() method

- Whenever we are printing any object reference internally __str__() method will be called which is returns string in the following format <__main__.classname object at 0x022144B0>
- To return meaningful string representation we have to override __str__() method.

```
class Student:
    def __init__(self,name,rollno):
        self.name=name
        self.rollno=rollno
    def __str__(self):
        return 'This is Student with Name:{} and Rollno:{}'.
                                           format(self.name,self
                                           .rollno)
>> s1=Student('John',101)
>> s2=Student('Ravi',102)
>> print(s1)
>> print(s2)
```

```
    '''Banking Application'''
class Account:
    def __init__(self,name,balance,min_balance):
        self.name=name
        self.balance=balance
        self.min_balance=min_balance

    def deposit(self,amount):
        self.balance +=amount

    def withdraw(self,amount):
        if self.balance-amount >= self.min_balance:
            self.balance -=amount
        else:
            print("Sorry, Insufficient Funds")

    def printStatement(self):
        print("Account Balance:",self.balance)

class Current(Account):
    def __init__(self,name,balance):
        super().__init__(name,balance,min_balance=-1000)
    def __str__(self):
        return "{}'s Current Account with Balance :{}".format(
                                           self.name,self.
                                           balance)

class Savings(Account):
    def __init__(self,name,balance):
        super().__init__(name,balance,min_balance=0)
    def __str__(self):
        return "{}'s Savings Account with Balance :{}".format(
                                           self.name,self.
                                           balance)

>> c=Savings("Durga",10000)
>> print(c)
>> c.deposit(5000)
>> c.printStatement()
>> c.withdraw(16000)
>> c.withdraw(15000)
>> print(c)

>> c2=Current('Ravi',20000)
>> c2.deposit(6000)
>> print(c2)
>> c2.withdraw(27000)
>> print(c2)
```

6.2 Magic or Special Methods

For every operator Magic Methods are available. To overload any operator we have to override that Method in our class. Internally + operator is implemented by using __add__() method.This method is called magic method for + operator. We have to override this method in our class.

```
''Program to overload + operator for our Book class objects'''
class Book:
    def __init__(self,pages):
        self.pages=pages

    def __add__(self,other):
        return self.pages+other.pages
>> b1=Book(100)
>> b2=Book(200)
print('The Total Number of Pages:',b1+b2)
```

```
The following is the list of operators and corresponding magic
                                   methods.
+ ---> object.__add__(self,other)
- ---> object.__sub__(self,other)
* ---> object.__mul__(self,other)
/ ---> object.__div__(self,other)
// ---> object.__floordiv__(self,other)
% ---> object.__mod__(self,other)
** ---> object.__pow__(self,other)
+= ---> object.__iadd__(self,other)
-= ---> object.__isub__(self,other)
*= ---> object.__imul__(self,other)
/= ---> object.__idiv__(self,other)
//= ---> object.__ifloordiv__(self,other)
%= ---> object.__imod__(self,other)
**= ---> object.__ipow__(self,other)
< ---> object.__lt__(self,other)
<= ---> object.__le__(self,other)
> ---> object.__gt__(self,other)
>= ---> object.__ge__(self,other)
== ---> object.__eq__(self,other)
!= ---> object.__ne__(self,other)
```

```
'''Overloading > and <= operators for Student class objects'''
class Student:

    def __init__(self,name,marks):
        self.name=name
        self.marks=marks

    def __gt__(self,other):
        return self.marks>other.marks

    def __le__(self,other):
        return self.marks<=other.marks

>> print("10>20 =",10>20)
>> s1=Student("John",100)
>> s2=Student("Ravi",200)
>> print("s1>s2=",s1>s2)
>> print("s1<s2=",s1<s2)
>> print("s1<=s2=",s1<=s2)
>> print("s1>=s2=",s1>=s2)
```

```
 '''Program to overload multiplication operator to work on Employee
                                objects'''
class Employee:

    def __init__(self,name,salary):
        self.name=name
        self.salary=salary
    def __mul__(self,other):
        return self.salary*other.days

class TimeSheet:
    def __init__(self,name,days):
        self.name=name
        self.days=days

>> e=Employee('John',500)
>> t=TimeSheet('John',25)
>> print('This Month Salary:',e*t)
```

Lab 5 Exercises

The objectives of this lab

- Understand the concept of polymorphism and its importance in Python.
- Apply polymorphism to solve real-world programming problems.

Lab 5 Assignments

1. Create a Python program that calculates the area of different shapes (circle, rectangle, triangle) using polymorphism.
2. Develop a Python program that represents different animals and their sounds using polymorphism.
3. Implement a Python program to calculate payments for different types of employees (hourly, salaried) using polymorphism.
4. Extend Problem 1 with error handling to handle invalid inputs gracefully.
5. Create a Python program to simulate multimedia players that can play different types of media files (audio, video) using polymorphism.