Python Programming

RGM College of Engineering & Technology (Autonomous)

Department of Computer Science & Engineering

AY:2021-2022

PYTHON'S OBJECT ORIENTED PROGRAMMING - 6

Guido Van Rossum

Agenda:

1. Types of Methods
2. Accessing Members of one class inside another class
3. Nested Methods
4. Garbage Collection

1. Types of Methods

Inside Python class 3 types of methods are allowed:

1. **Instance Methods**
2. **Class Methods**
3. **Static Methods**

1. Instance Methods:

Definition:

- Inside method implementation if we are using any instance variables then this method always talks about particular object. Such type of methods are called instance methods.
- The first argument to the instance method always self, which is reference variable to the current object. Using this self, we can access the values of instance variables.
- Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

Demo Program:

```
class student:
    def __init__(self,name,marks):
        self.name = name
        self.marks = marks

    def display(self):
        print("Hi ",self.name)
        print("Your Marks are :",self.marks)

    def grade(self):
        if self.marks >= 60:
            print("You got First Grade")
        elif self. marks >=50:
            print("You got Second Grade")
        elif self.marks >=35:
            print("You got Third Grade")
        else:
            print("You are Failed...")

n=int(input('Enter number of students:'))

for i in range(n):
    name = input('Enter Name:')
    marks = int(input('Enter Marks:'))
    s = student(name,marks)
    s.display()
    s.grade()
    print()
```

```
Enter number of students:4
Enter Name:Sourav
Enter Marks:67
Hi  Sourav
Your Marks are : 67
You got First Grade

Enter Name:Rahul
Enter Marks:56
Hi  Rahul
Your Marks are : 56
You got Second Grade

Enter Name:Sachin
Enter Marks:45
Hi  Sachin
Your Marks are : 45
You got Third Grade

Enter Name:Karthikeya
Enter Marks:7
Hi  Karthikeya
Your Marks are : 7
You are Failed...
```

Setter and Getter Methods

We can set and get the values of instance variables by using setter and getter methods.

i. Setter Methods:

- setter methods can be used to set values to the instance variables.
- setter methods also known as mutator methods.

Syntax:

```
def setVariableName(self,variable):
    self.variable=variable
```

Eg:

```
def setMarks(self,marks):
    self.marks = marks
```

Note: As per coding standards, this syntax is fixed.

ii. Getter Methods:

- Getter methods can be used to get values of the instance variables.
- Getter methods also known as accessor methods.

Syntax:

```
def getVariable(self):
    return self.variable
```

Eg:

```
def getMarks(self):
    return self.marks
```

Note:

Both setter methods and getter methods are the best examples of Instance Methods.

Demo Program on Setter and Getter Methods

```
class Student:

    def setName(self,name):
        self.name = name

    def getName(self):
        return self.name

    def setMarks(self,marks):
        self.marks=marks

    def getMarks(self):
        return self.marks

n=int(input('Enter number of students:'))
students = []
for i in range(n):
    s=Student()
    name=input('Enter Name:')
    s.setName(name)
    marks=int(input('Enter Marks:'))
    s.setMarks(marks)
    students.append(s)

for s in students:
    print('Hello ',s.getName())
    print('Your Marks are:',s.getMarks())
    print()
```

```
Enter number of students:4
Enter Name:Karthi
Enter Marks:56
Enter Name:saha
Enter Marks:88
Enter Name:Nikki
Enter Marks:76
Enter Name:cherry
Enter Marks:77
Hello  Karthi
Your Marks are: 56

Hello  saha
Your Marks are: 88

Hello  Nikki
Your Marks are: 76

Hello  cherry
Your Marks are: 77
```

```
class Student:

    def setName(self,name):
        self.name = name

    def getName(self):
        return self.name

    def setMarks(self,marks):
        self.marks=marks

    def getMarks(self):
        return self.marks

n=int(input('Enter number of students:'))
students = []
for i in range(n):
    s=Student()
    name=input('Enter Name:')
    s.setName(name)
    marks=int(input('Enter Marks:'))
    s.setMarks(marks)
    students.append(s)

for s in students:
    print()
    print('Hello ',s.getName())
    print('Your Marks are:',s.getMarks())
    print()
```

```
Enter number of students:3
Enter Name:ram
Enter Marks:44
Enter Name:raj
Enter Marks:66
Enter Name:rocky
Enter Marks:45

Hello  ram
Your Marks are: 44


Hello  raj
Your Marks are: 66


Hello  rocky
Your Marks are: 45
```

2. Class Methods

- Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.
- We can declare class method explicitly by using @classmethod decorator.
- For class method, we should provide cls variable at the time of declaration.
- We can call class method by using either class name or object reference variable.

Eg:

```
class bird:
    wings = 2

    @classmethod
    def fly(cls,name):
        print('{} flying with {} wings'.format(name,cls.wings))
        # We are not using any instance variables

bird.fly('Parrot')
bird.fly('Eagle')
```

```
Parrot flying with 2 wings
Eagle flying with 2 wings
```

Python Program to track the number of objects created for a class.

I.

```
class Test:
    count=0                 # Class level or static variable
    def __init__(self):
        Test.count =Test.count+1

    @classmethod
    def getnoOfObjects(cls):
        print('The number of objects created for test class:',cls.count)

Test.getnoOfObjects()
```

```
The number of objects created for test class: 0
```

II.

```
class Test:
    count=0                 # Class level or static variable
    def __init__(self):
        Test.count =Test.count+1

    @classmethod
    def getnoOfObjects(cls):
        print('The number of objects created for test class:',cls.count)

Test.getnoOfObjects()
t1=Test()
t2=Test()
t3=Test()
t4=Test()
t5=Test()
Test.getnoOfObjects()
```

```
The number of objects created for test class: 0
The number of objects created for test class: 5
```

III.

```
class Test:
    count=0
    def __init__(self):
        Test.count =Test.count+1

    @classmethod
    def noOfObjects(cls):
        print('The number of objects created for test class:',cls.count)

t1=Test()
t2=Test()
Test.noOfObjects()
t3 = Test()
```

```
The number of objects created for test class: 2
```

IV.

```
class Test:
    count=0
    def __init__(self):
        Test.count =Test.count+1

    @classmethod
    def noOfObjects(cls):
        print('The number of objects created for test class:',cls.count)

t1=Test()
t2=Test()
t3 = Test()
Test.noOfObjects()
```

```
The number of objects created for test class: 3
```

Instance Method	Class Method
1. Inside method body if we are using at least one instance variable then compulsory we should declare that method as instance method.	1. Inside method body if we are using only static variables and if we are not using instance variables then we have to declare that method as class method.
2. Inside instance method we can access both instance and static variables.	2. Inside classmethod we can access only static variables and we cannot access instance variables
3. To declare instance method we are not required to use any decorator.	3. To declare class method compulsory we should use @classmethod decorator
4. The first argument to the instance method should be self, which is reference to current object and by using self, we can access instance variables inside method.	4. The first argument to the classmethod should be cls, which is reference variable current class object and by using that we can access static variables
5. We can call instance method by using object reference.	5. We can call classmethod either by using object reference or by using class name,but recommended to use classname.

3. Static Methods

- In general these methods are general utility methods.
- Inside these methods we won't use any instance or class variables.
- Here we won't provide self or cls arguments at the time of declaration.
- We can declare static method explicitly by using @staticmethod decorator.
- We can access static methods by using class name or object reference. But recommended to use class name.

Demo Program on static method:

```
class RgmMath:

    @staticmethod
    def add(a,b):
        print("The Sum : ", a + b)

    @staticmethod
    def product(a,b):
        print('The Product:',a*b)

    @staticmethod
    def average(a,b):
        print('The average:',(a+b)/2)

RgmMath.add(10,20)
RgmMath.product(10,20)
RgmMath.average(10,20)
```

```
The Sum :  30
The Product: 200
The average: 15.0
```

Note:

- In general we can use only instance and static methods.
- Inside static method we can access class level variables by using class name.
- class methods are most rarely used methods in python.

Instance Method vs Class Method vs Static Method

1. If we are using any instance variables inside method body then we should go for instance method. We should call by using object reference only.
2. If we are using only static variables inside method body then this method no way related to a particular object, we should declare such type of methods as classmethod. We can declare class method explicitly using @classmethod decorator. We can call either by using object reference or by using class name.
3. If we are not using any instance variable and any static variable inside method body, to define such type general utility methods we should go for static methods. We can declare static methods explicitly by using @staticmethod decorator. We can call either by using object reference or by using class name.

Key Points:

- If we are using only Instance Variables --> instance method
- If we are using only Static Variables --> class method
- If we are using Instance Variables and Static Variables --> instance method
- If we are using instance variables and local variables --> instance method
- If we are using static variables and local variables --> class method
- If we are using local variables --> static method

If we are not using any decorator???!

- For class method, @classmethod decorator is mandatory.
- For static methods, @staticmethod decorator is optional.
- Without any decorator the method can be either instance method or static method.
- If we are calling any method by using object reference then it is treated as instance method.
- If we are calling by using class name then it is treated as static method.

Example Programs:

I.

```
class Test:
    def m1():
        print('Some Method')

t = Test()
t.m1()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-dcf377dae7d6> in <module>
      4
      5 t = Test()
----> 6 t.m1()

TypeError: m1() takes 0 positional arguments but 1 was given
```

Explanation:

- If a method without any decorator, it may be either Instance method or Static method.
- In the above example, we are calling the method 'm1' with object reference, so PVM decides that 'm1' is Instance method. If it is the Instance method, compulsory the first argument is 'self'. But for the method ‘m1’ self is not there. So, Immediately we will get an error.

TypeError: m1() takes 0 positional arguments but 1 was given.

- Reason for the above error is, PVM provides self value to the instance method, but in the above example method 'm1' doesn't take any argument. So it gives TypeError.

II.

```
class Test:
    def m1(x):  # Self value given by PVM can store in x (We can give any name)
        print("Some Method")
t=Test()                          # Valid code
t.m1()
```

```
Some Method
```

III.

```
class Test:
    def m1(x):  # Self value given by PVM can store in x (We can give any name)
        print("Some Method")
t=Test()                              # InValid code
t.m1(10)   # self value & 10 are passing.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-a104074401b6> in <module>
      3         print("Some Method")
      4 t=Test()                              # InValid code
----> 5 t.m1(10)   # self value & 10 are passing.

TypeError: m1() takes 1 positional argument but 2 were given
```

IV.

```
class Test:
    def m1():
        print("Some Method")

Test.m1()                    # in this case, 'm1' is static method

                    # Because, we are calling 'm1' with class name
```

```
Some Method
```

V.

```
class Test:
    def m1(x):
        print("Some Method")
Test.m1()  # Invalid, m1 expects 1 argument, but we are not providing
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-6a38158f2852> in <module>
      2     def m1(x):
      3         print("Some Method")
----> 4 Test.m1()

TypeError: m1() missing 1 required positional argument: 'x'
```

VI.

```
class Test:
    def m1(x):
        print("Some Method")
Test.m1(10)                    # Valid code
```

```
Some Method
```

2. Accessing Members of one class inside another class

- We can access members of one class inside another class based on our requirement.

Eg:

```
class employee:

    def __init__(self,eno,ename,esal):
        self.eno = eno
        self.ename = ename
        self.esal = esal

    def display(self):
        print('Employee Number:',self.eno)
        print('Employee Name:',self.ename)
        print('Employee Salary:',self.esal)


class Manager:

    def updateempsal(emp):
        emp.esal=emp.esal+10000
        # Manager class accessing esal property from employee class
        emp.display()
        # Manager class accessing display method from employee class

emp = employee(111,'Karthi',47800)
Manager.updateempsal(emp)
```

```
Employee Number: 111
Employee Name: Karthi
Employee Salary: 57800
```

Inner Classes:

- Sometimes we can declare class inside a class, such type of classes are called **inner classes.**
- Without existing one type of object, if there is no chance of existing another type of object, then we should go for inner classes.

Consider the following examples,

Eg 1:

- Without existing University object, there is no chance of existing Department object. Department should be the part of university and cannot exist alone. Hence we should declare Department class inside University class.

```
class University:
    class Department:
        pass
```

In the above code, **University class** is outer class and **Department class** is Inner class.

Eg 2:

- Without existing Car object, there is no chance of existing Engine object. Engine should be part of the Car, Hence we should declare Engine class inside Car class.

```
class car:
    class Engine:
        pass
```

In the above code, Car class is outer class where as Engine class is Inner class.

Eg 3:

- Without existing Head, there is no chance of existing Brain. Hence Brain class should be defined inside Head class.

```
class Head:
    class Bain:
        pass
```

In the above code, Head class is outer class where as Brain class is inner class.

Note:

- Without existing outer class object there is no chance of existing inner class object. Hence inner class object is always associated with outer class object.

Advantages of Inner Classes:

1. Inner classes improves modularity of the application.

2. Inner classes improves security of the application.

Demo Programs to define and use of Inner classes:

I.

```
class Outer:

    def __init__(self):
        print("Outer class object creation")

    class Inner:
        def __init__(self):
            print("Inner class object creation")

        def m1(self):
            print("Inner class method")

o = Outer()
i = Inner()  # You can't create inner class object directly
i.m1()
```

```
Outer class object creation
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-039c30d790d0> in <module>
     12
     13 o = Outer()
---> 14 i = Inner()
     15 i.m1()

NameError: name 'Inner' is not defined
```

II.

```
class Outer:

    def __init__(self):
        print("Outer class object creation")

    class Inner:
        def __init__(self):
            print("Inner class object creation")

        def m1(self):
            print("Inner class method")

o = Outer()
i = o.Inner()
i.m1()
```

```
Outer class object creation
Inner class object creation
Inner class method
```

The following are various possible syntaxes for calling inner class method.

i)

```
o=Outer()
i=o.Inner()
i.m1()
```

ii)

```
i=Outer().Inner()
i.m1()
```

iii)

```
Outer().Inner().m1()
```

III.

```
class Outer:

    def __init__(self):
        print("Outer class object creation")

    class Inner:
        def __init__(self):
            print("Inner class object creation")

        def m1(self):
            print("Inner class method")

i = Outer().Inner()
i.m1()
```

```
Outer class object creation
Inner class object creation
Inner class method
```

IV.

```
class Outer:

    def __init__(self):
        print("Outer class object creation")

    class Inner:
        def __init__(self):
            print("Inner class object creation")

        def m1(self):
            print("Inner class method")

Outer().Inner().m1()
```

```
Outer class object creation
Inner class object creation
Inner class method
```

Nesting of Inner Classes Demo Programs

I.

```
class Outer:

    def __init__(self):
        print("Outer class object creation")

    class Inner:
        def __init__(self):
            print("Inner class object creation")

        class InnerInner:
            def __init__(self):
                print("InnerInner class object creation")

            def m1(self):
                print("Nested Inner class method")
```

```
o = Outer()
i = o.Inner()
ii = i.InnerInner()
ii.m1()
```

```
Outer class object creation
Inner class object creation
InnerInner class object creation
Nested Inner class method
```

II.

```
class Outer:

    def __init__(self):
        print("Outer class object creation")

    class Inner:
        def __init__(self):
            print("Inner class object creation")

        class InnerInner:
            def __init__(self):
                print("InnerInner class object creation")

            def m1(self):
                print("Nested Inner class method")

Outer().Inner().InnerInner().m1() # Short cut approach
```

```
Outer class object creation
Inner class object creation
InnerInner class object creation
Nested Inner class method
```

Nested class may contains static methods also. Consider the following example,

III.

```
class Outer:

    def __init__(self):
        print("Outer class object creation")

    class Inner:
        def __init__(self):
            print("Inner class object creation")

        class InnerInner:
            def __init__(self):
                print("InnerInner class object creation")

            @staticmethod
            def m1():
                print("Nested Inner class method")

Outer().Inner().InnerInner().m1() # Static method called using it's object reference
```

```
Outer class object creation
Inner class object creation
InnerInner class object creation
Nested Inner class method
```

IV.

```
class Outer:

    def __init__(self):
        print("Outer class object creation")

    class Inner:
        def __init__(self):
            print("Inner class object creation")

        class InnerInner:
            def __init__(self):
                print("InnerInner class object creation")

            @staticmethod
            def m1():
                print("Nested Inner class method")

Outer().Inner().InnerInner.m1() # Static method called using it's class name
```

```
Outer class object creation
Inner class object creation
Nested Inner class method
```

Inside a class we can declare any number of inner classes.

V.

```
class Human:
    class Head:
        def talk(self):
            print("Talking ")

        class Brain:
            def think(self):
                print("Thinking ")

human = Human()
head = human.Head()
head.talk()
brain = head.Brain()
brain.think()
```

```
Talking
Thinking
```

Suppose your requirement is,

- Whenever if you are created Human object, automatically Head object should be create.
- Whenever if you are created Head object, automatically Brain object should be create.

VI.

```
class Human:

    def __init__(self,name):
        print("Human Object Creation...")
        self.name = name
        self.head = self. Head()
        self.brain = self


    class Head:

        def __init__(self):
            print("Head Object Creation...")
            self.brain = self.Brain()


        class Brain:
            def __init__(self):
                print("Brain Object Creation...")

human = Human("Karthi")
```

```
Human Object Creation...
Head Object Creation...
Brain Object Creation...
```

VII.

```
class Human:

    def __init__(self,name):
        print("Human Object Creation...")
        self.name = name
        self.head = self. Head()
        self.brain = self

    def info(self):
        print("Hello myself :",self.name)
        print('I am full busy with ')
        self.head.talk()
        self.head.brain.think()


    class Head:

        def __init__(self):
            print("Head Object Creation...")
            self.brain = self.Brain()

        def talk(self):
            print("Talking... ")

        class Brain:

            def __init__(self):
                print("Brain Object Creation...")

            def think(self):
                print("Thinking... ")

human = Human("Karthi")
human.info()
```

```
Human Object Creation...
Head Object Creation...
Brain Object Creation...
Hello myself : Karthi
I am full busy with
Talking...
Thinking...
```

VIII.

```
class Person:

    def __init__(self):
        print("Person Object Creation...")
        self.dob = self.DOB()

    class DOB:

        def __init__(self):
            print("DOB object creation...")

p = Person()   # both objects will be created
```

```
Person Object Creation...
DOB object creation...
```

IX.

```
class Person:

    def __init__(self,name,dd,mm,yyyy):
        print("Person Object Creation...")
        self.name = name
        self.dob = self.DOB(dd,mm,yyyy)  # DOB object cretaed with provided data

    def info(self):
        print("Name : ",self.name)
        self.dob.display()

    class DOB:

        def __init__(self,dd,mm,yyyy):
            print("DOB object creation...")
            self.dd = dd
            self.mm = mm
            self.yyyy = yyyy

        def display(self):
            print("DOB = {}/{}/{}".format(self.dd,self.mm,self.yyyy))

p = Person("Karthi",26,6,2014)
p.info()
```

```
Person Object Creation...
DOB object creation...
Name :  Karthi
DOB = 26/6/2014
```

3. Nested Methods

- We can declare a method inside another method, such type of methods are called Nested Methods.
- Inside a method if any functionality repeatedly required, that functionality we can define as a separate method and we can call that method any number of times based our requirement.

Demo Program on Nested Methods - I

```
class Test:

    def m1(self):

        def calc(a,b):
            print('The sum is :',a+b)
            print('The Difference is :',a-b)
            print("The Product is : ",a*b)
            print("The Average is :",(a+b)/2)

        calc(10,20)

t = Test()
t.m1()
```

```
The sum is : 30
The Difference is : -10
The Product is :  200
The Average is : 15.0
```

Demo Program on Nested Methods - II

```
class Test:

    def m1(self):

        def calc(a,b):
            print('The sum is :',a+b)
            print('The Difference is :',a-b)
            print("The Product is : ",a*b)
            print("The Average is :",(a+b)/2)
            print()

        calc(10,20)
        calc(101,200)
        calc(1000,2000)
        calc(1050,2050)

t = Test()
t.m1()
```

```
The sum is : 30
The Difference is : -10
The Product is :  200
The Average is : 15.0

The sum is : 301
The Difference is : -99
The Product is :  20200
The Average is : 150.5

The sum is : 3000
The Difference is : -1000
The Product is :  2000000
The Average is : 1500.0

The sum is : 3100
The Difference is : -1000
The Product is :  2152500
The Average is : 1550.0
```

Advantages of Nested Methods:

1. Code Reusability
2. Modularity of the application will be improved.

4. Garbage Collection

- In Languages like C++, Programmer is responsible for both creation and destruction of objects.
- Usually the programmer taking very much care while creating objects, but neglecting destruction of useless objects.
- Because of this negligence, total memory may filled with useless objects which creates memory problems and total application will be down with these memory problems.
- But in Java (or) Python, we have some assistant which is always running in the background to destroy useless objects.
- Because of this assistant, the chance of failing Python program with memory problems is very less. This assistant is nothing but Garbage Collector.

Note: The main objective of Garbage Collector is to destroy useless objects.

When we say an object is eligible for Garbage Collection?

- If the object does not contain any reference variable then only it is eligible for Garbage Collection.
- If the reference count is zero then only object eligible for Garbage Collection.

How to enable and disable Garbage Collector in our program?

- Based on our requirement, we can enable or disable the Garbage Collector.
- By default, Garbage Collector is enabled.
- If you don't want Garbage Collector, you can disable it.
- For all these activities (i.e., to enable, to disable garbage collector, to check whether it is enabled or not), one special module we have to use (Of course it is Python's built-in module).This module is gc module.

In this context we can use the following functions of gc module.

1. gc.isenabled(): ===> Returns True if Garbage Collector is enabled
2. gc.disable(): ===> To disable Garbage Collector explicitly
3. gc.enable(): ===> To enable Garbage Collector explicitly

Demo program on Garbage Collector – I:

```
import gc
print(gc.isenabled())     # True
gc.disable()
print(gc.isenabled())     # False
gc.enable()
print(gc.isenabled())     # True
```

```
True
False
True
```

In What situations, we will go for disabling Garbage Collector explicitly?

- Suppose, if you feel in your application, memory problems won't be raised.
- If you are going to feel that you are creating very less number of arguments.
- In the above cases, why unnecessarily to run the Garbage collector at background (because of this, performance is going to be down).
- That's why in these cases we will disable the Garbage Collector.

Any question?

If you try to practice programs yourself, then you will learn many things automatically

Spend few minutes and then enjoy the study

Thank You