Course Title: Programming Language II

Course Code: CSE 111

Semester: Summer 2020

Topic: Object-Oriented Programming (Encapsulation & Operator Overloading)

Table of Contents

1. Python - public, private and protected:

Object-oriented languages control the access to class resources by public, private and protected keywords.

In python, it doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with single or double underscore to emulate the behavior of protected and private access specifiers.

Before starting python encapsulation, let's discuss how python handle public, private and protected members-

1.1. Public member:

Public members (generally methods declared in a class) are accessible from outside the class. The object of the same class is required to invoke a public method. All members in a Python class are public by default. Any member can be accessed from outside the class environment.

Here, in the following example, variables can call outside of call and also can change the variables.

```python
class Student:
    def __init__(self, name, id):
        self.name=name
        self.id=id
s1=Student("Kiran",18101)
print(s1.id)
s1.id=18202
print(s1.id)
```

Output:

```
18101

18202
```

1.2. Protected member

Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it. This enables specific resources of the parent class to be inherited by the child class.

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. In fact, this doesn't prevent instance variables from accessing or modifying the instance.

Here in following example, it can still perform the following operations and accessing and modifying instance variables prefixed with _ from outside its class.

```
class Student:
    def __init__(self, name, id):
        self._name=name # protected attribute
        self._id=id     # protected attribute
s1=Student("Kiran",18101)
print(s1._id)
s1._id=18202
print(s1._id)
```

Output:

```
18101

18202
```

1.3. Private member:

Private members of a class are denied access from the environment outside the class. They can be handled only from within the class.

Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class. In Python, there is no existence of Private instance variables that cannot be accessed except inside a class. However, to define a private member prefix the member name with double underscore "__" (double underscore).

Here in following example, attribute cannot be change outside of the class as it is assigned as private attribute.

```
class Student:

    def __init__(self):
        self.__id = 18101       # private attribute
        self.__name = "Kiran"   # private attribute

    def info(self):
        print(self.__id)

s1 = Student()
s1.info()
s1.__id = 18202
# will not change variable because its private
s1.info()
```

Output:

```
18101

18101
```

Also from the following example we can differentiate public, private and protected attributes. Here public and protected attribute can access outside the class but in case of private attribute it shows an error.

```
class Robot(object):
   def __init__(self):
      self.a = 123   #public attribute
      self._b = 123  #protected attribute
      self.__c = 123 #private attribute

obj = Robot()
print(obj.a)
print(obj._b)
print(obj.__c)
```

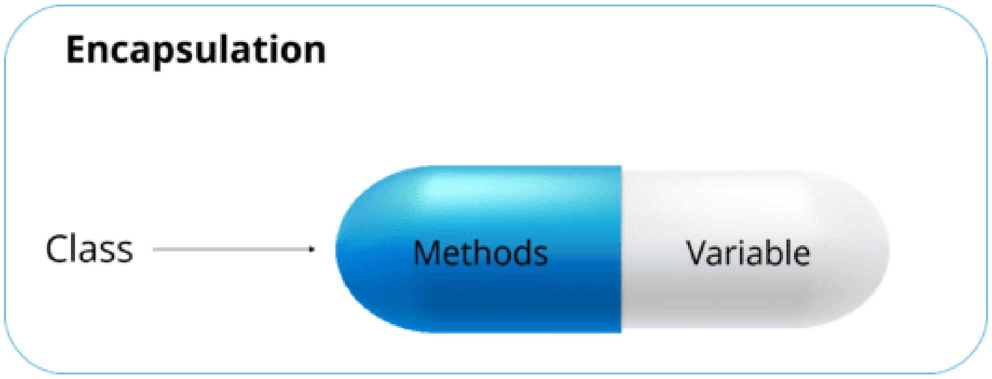
Output:

```
123
123
Traceback (most recent call last):
  File "test.py", line 10, in &lt;module&gt;
    print(obj.__c)
AttributeError: 'Robot' object has no attribute '__c'
```

2. <u>**Encapsulation:**</u>

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those type of variables are known as **private variable**.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



Example:

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”. As using encapsulation also hides the data. In this example, the data of any of the sections like sales, finance or accounts are hidden from any other section. So encapsulation provides security by hiding the data from the outside world.

In Python, Encapsulation can be achieved by declaring the data members of a class either as **private** or **protected**.

Let us see how access modifiers help in achieving Encapsulation.

2.1 Encapsulation Using Private Members:

If we declare any variable or method as private, then they can be accessed only within the class in which they are defined.
In the below example, 'length' and 'breadth' are the two private variables declared and can be accessed within the class 'Rectangle'.

```
class Rectangle:
  __length = 0 #private variable
  __breadth = 0#private variable
  def __init__(self): #constructor
      self.__length = 5
      self.__breadth = 3
#printing values of the private variable within the class
      print(self.__length)
      print(self.__breadth)

rec = Rectangle() #object created for the class 'Rectangle'
#printing values of the private variable outside the class using
the object created for the class 'Rectangle'
print(rec.length)
print(rec.breadth)
```

Output:

```
5

3

Traceback (most recent call last):

File "main.py", line 11, in <module>

print(rec.__length)

AttributeError: 'Rectangle' object has no attribute '__length'
```

Since we have accessed private variables in the main() method, i.e., outside the class 'Rectangle', we got an error. Hence in the above program, Encapsulation is achieved using the private variables 'length' and 'breadth'.

2.2 Encapsulation Using Protected Members:

Protected members can be accessed within the class in which they are defined and also within the derived classes.

In the below example, 'length' and 'breadth' are the two protected variables defined inside the class 'Shape'.

```
class Shape:
    _length = 10  #protected variables
    _breadth = 20 #protected variables

class Circle(Shape):
  def __init__(self):
    #printing protected variables in the derived class
    print(self._length)
    print(self._breadth)

cr = Circle()
#printing protected variablesoutsidethe class 'Shape' in which they
 are defined
print(cr.length)
print(cr.breadth)
```

Output:

```
10

20

Traceback (most recent call last):

File "main.py", line 11, in <module>

print(cr.length)

AttributeError: 'Circle' object has no attribute 'length'
```

When we try to access protected variables in the derived class, we got output. But, in the main() method, we got an error. Hence in the above example, Encapsulation is achieved using the protected variables 'length' and 'breadth'.

3. <u>Operator Overloading:</u>

Python operators work for built-in classes. But the same operator behaves differently with different types.

For example, the " +" operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

Output:

```
Traceback (most recent call last):

  File "<string>", line 9, in
<module>

    print(p1+p2)

TypeError: unsupported operand
type(s) for +: 'Point' and 'Point'
```

Here, we can see that a “Type Error” was raised, since Python didn't know how to add two Point objects together. However, we can achieve this task in Python through operator overloading. But before that let’s get a notion about special functions.

Python Special Function:

Special functions in python are the functions which are used to perform special tasks. These special functions have __(double underscore) as prefix and suffix to their name as we see in __init__() method which is also a special function

	Name	**Symbol**	**Special Function**
Mathematical Operator	Addition	`+`	`__add__(self, other)`
	Subtraction	`-`	`__sub__(self, other)`
	Division	`/`	`__truediv__(self, other)`
	Floor Division	`//`	`__floordiv__(self, other)`
	Modulus(or Remainder)	`%`	`__mod__(self, other)`
	Power	`**`	`__pow__(self, other)`
Assignment Operator	Increment	`+=`	`__iadd__(self, other)`
	Decrement	`-=`	`__isub__(self, other)`
	Product	`*=`	`__imul__(self, other)`
	Division	`/=`	`__idiv__(self, other)`
	Modulus	`%=`	`__imod__(self, other)`
	Power	`**=`	`__ipow__(self, other)`
Relational Operator	Less than	`<`	`__lt__(self, other)`
	Greater than	`>`	`__gt__(self, other)`
	Equal to	`==`	`__eq__(self, other)`
	Not equal	`!=`	`__ne__(self, other)`
	Less than or equal to	`<=`	`__le__(self, other)`
	Greater than or equal to	`> =`	`__gt__(self, other)`

Example:

To overload the "+" operator, we will need to implement __add__() function in the class. In the below code example we will overload the + operator.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)
```

Output:

```
(3, 5)
```

In the program above, __add__() is used to overload the + operator i.e. when + operator is used with two Point class objects then the function __add__() is called.

__str__() is another special function which is used to provide a format of the object that is suitable for printing. when we use p1 + p2, Python calls p1.__add__(p2) which in turn is Point.__add__(p1,p2). After this, the addition operation is carried out the way we specified.