جامعة الانبار

كلية علوم الحاسوب وتكنولوجيا المعلومات

قسم أنظمة شبكات الحاسوب

برمجة كيانية OOP

المرحلة الثانية

الفصل الدراسي الأول والثاني

مدرس المادة

م.د. سميه عبدالله حمد

Dr. Sumaya Abdulla Hamad ٢٠٢٢ -٢٠٢٣

Python - Public, Protected, Private Members

Classical object-oriented languages, such as C++ and Java, control the access to class resources by public, private, and protected keywords. Private members of the class are denied access from the environment outside the class. They can be handled only from within the class.

Public Members

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. This arrangement of private instance variables and public methods ensures the principle of data encapsulation.

All members in a Python class are **public** by default. Any member can be accessed from outside the class environment.

Example: Public Attributes

```
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self, name, age):
        self.name=name # instance attribute
        self.age=age # instance attribute
```

You can access the Student class's attributes and also modify their values, as shown below.

Example: Access Public Members

```
>>> std = Student("Steve", 25)
>>> std.schoolName
'XYZ School'
>>> std.name
'Steve'
>>> std.age = 20
>>> std.age
20
```

Protected Members

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 from being accessed unless it is from within a sub-class.

Example: Protected Attributes

```
class Student:
    _schoolName = 'XYZ School' # protected class attribute

    def __init__(self, name, age):
        self._name=name  # protected instance attribute
        self._age=age # protected instance attribute
```

In fact, this doesn't prevent instance variables from accessing or modifying the instance. You can still perform the following operations:

Example: Access Protected Members

```
>>> std = Student("Swati", 25)
>>> std._name
'Swati'
>>> std._name = 'Dipa'
>>> std._name
'Dipa'
```

However, you can define a property using property decorator and make it protected, as shown below.

Example: Protected Attributes

```
class Student:
    def __init__(self,name):
            self._name = name
    @property
    def name(self):
            return self._name
    @name.setter
    def name(self,newname):
            self._name = newname
```

Above, @property decorator is used to make the name() method as property and @name.setter decorator to another overloads of the name() method as property setter method. Now, _name is protected.

Example: Access Protected Members

```
>>> std = Student("Swati")
>>> std.name
'Swati'
>>> std.name = 'Dipa'
>>> std.name
'Dipa'
>>> std._name # still accessible
```

Above, we used std.name property to modify _name attribute. However, it is still accessible in Python. Hence, the responsible programmer would refrain from accessing and modifying instance variables prefixed with _ from outside its class.

Private Members

Python 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 a single or double underscore to emulate the behavior of protected and private access specifiers.

The double underscore __ prefixed to a variable makes it **private**. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:

Example: Private Attributes

```
class Student:
    __schoolName = 'XYZ School' # private class attribute

    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__salary=age # private instance attribute
    def __display(self):  # private method
         print('This is private method.')
```

Example:

```
>>> std = Student("Bill", 25)
>>> std.__schoolName
AttributeError: 'Student' object has no attribute '__schoolName'
>>> std.__name
AttributeError: 'Student' object has no attribute '__name'
>>> std.__display()
AttributeError: 'Student' object has no attribute '__display'
```

Python performs name mangling of private variables. Every member with a double underscore will be changed to _object._class__variable. So, it can still be accessed from outside the class, but the practice should be refrained.

Example:

```
>>> std = Student("Bill", 25)
>>> std._Student__name
'Bill'
>>> std._Student__name = 'Steve'
>>> std._Student__name
'Steve'
>>> std._Student__display()
'This is private method.'
```

Thus, Python provides conceptual implementation of public, protected, and private access modifiers, but not like other languages like C#, Java, C++.