2.2 OOP Principles I

Object-oriented programming (OOP) – OOP Principles I

There are four fundamental Object-Oriented Programming (OOP) Principles - encapsulation, inheritance, abstraction, and polymorphism.

Encapsulation

Encapsulation refers to the idea that a class's internal logic or implementation details are hidden from the rest of the code. In other words, the logic you have written inside a method cannot be seen from outside that method when the program is running. All that can be done is call that method using its method signature. Whatever can be accessed from outside a class is called the class's public interface.

In Python, there is no concept of private variables. By convention, we use an underscore in front of an attribute (an attribute with a leading underscore) if that attribute is not expected to be referenced from outside the class. Note that even the attributes with a leading underscore can be accessed anywhere in the code. However, it is not a good programming practice to access attributes with a leading underscore from outside their class.

Pythonic way to expose these private attributes to the outside is to use “@property” decorator. A getter returns the value of an attribute. The setter sets the value of an attribute. The method with the "@property" decorator, also known as the getter, has the same name as the private attribute without the leading underscore. It is used to access the value of the private attribute outside of the class. To set the attribute's value, we define another method with the same name as the private attribute without the leading underscore and use the decorator @<private_attribute>. setter, where private_attribute is an attribute name with a leading underscore. Using such getters and setters, we can add validation logic around getting and setting the value of the private attribute. If the attribute was directly accessed without a getter and setter, such validation rules could not be exposed.

In summary, the main benefit of using encapsulation in Python is that it allows us to add constraints and logic when accessing and setting values of private attributes. Also, we can change the implementation of a method without changing the way this method is accessed from other places, as long as the method's signature does not change. This helps to keep our code simple and easy to maintain. In other words, we can change one place of the code without having a significant impact on the rest of the code.

Inheritance

Inheritance refers to the subclass superclass relationship. A subclass can also be referred to as a child class or derived class. Similarly, the superclass is also referred to as parent class or base class. Subclass-superclass relationships are in abundance around us. For example, if we consider birds, we have ducks, cuckoos, etc. They are all birds, having some common properties, such as two legs and a beak. However, different bird types may have specific attributes and methods on top of these common properties. For example, a duck's sound is different from the sound of a cuckoo. As another example, consider different person types. If we take an employee and a baseball player, they both have common properties, such as the name. However, they have attributes and methods specific to them as well. For example, a baseball player is good at a sport, while the employee works in some office. Another example is the bicycle. All the bikes have a set of common properties and methods, but they may have some type-specific attributes and methods as well.

With inheritance, we establish a connection, a relationship between the subclass and the superclass. This relationship is called an 'is-a relationship.' For example, we can say a 'dog is a(n) animal' or a 'super warrior is a warrior.' Note that this is different from the 'has a' relationship, which usually exists between an object and its attribute. Note that the 'has a' relationship can exist between two objects as well. For example, if we take a walking stick as another object, we can say 'a warrior has a walking stick'. Here the relationship is 'has a', not 'is a'. Hence, whenever you decide to make an inheritance relationship between two classes, see if an 'is a' relationship exists between the two classes. For example, it would not make sense to say, “a walking stick is a warrior.”

In python, a subclass can extend more than one superclass. It inherits all of the class and instance members of the superclasses. By members, we mean both attributes and methods. The inherited attributes can be used directly, just like any other attribute in the subclass. If you want, you can declare an attribute in the subclass with the same name as the one in the super-class, thus hiding it. You can also declare new fields in the subclass that are not in the superclass. This is essentially what we mean by the sub-class extending the superclass; the subclass will have all the superclass functionality and some more functionality.

Like attributes, inherited methods can be used in the subclass directly as they are. If you want, you can have a new method in the subclass with the same signature as the one in the superclass, thus overriding it. This is a perfectly valid implementation. You can, of course, declare new methods in the subclass that are not in the superclass.

As for the constructor, the sub-class can invoke the superclass's constructor by explicitly calling it using the super keyword. Even if you do not explicitly call it, it will be implicitly called. If superclass attributes have to be set, the subclass constructor MUST explicitly call the superclass constructor and pass relevant parameters.

Python does not support method overloading by default. This is because Python is an interpreted language. When we have multiple methods with the same name but different parameters, only the method defined lastly in the code is used. However, there are other ways to achieve method overloading. One approach is to use default values for the parameters. Note that method overloading and method overriding are different. In method overloading, the subclass has the same method signature as the superclass but has a different implementation. In method overloading, we consider a method inside a single class and change the way how it is called by the way we use the parameters that have been given a default value.

Since the superclass captures all the common members, we need to update only the code in the superclass whenever such a common member has to be changed. This means less code redundancy and easy maintenance of code.

◀ 2.1 Quiz

Jump to...

GET IN TOUCH

University of Moratuwa
Centre for Open & Distance Learning
CODL

011 308 2787/8

011 265 0301 ext. 3850,3851

open@uom.lk

University Website

CODL Website

Data retention summary

mora_logo.png Sponsored By logo.png