Chapter 13 Object Oriented Programming (OOP)

Contents

Syllabus Learning Outcomes

2.5 Fundamentals of Object-Oriented Programming
Understand concepts of encapsulation, inheritance, and polymorphism.
2.5.1 Define and understand classes and objects.
2.5.2 Understand encapsulation and how classes support information hiding and implementation independence.
2.5.3 Understand inheritance and how it promotes software reuse.
2.5.4 Understand polymorphism and how it enables code generalisation.
Exclude: method overloading and multiple inheritance

1 Components of Object-Oriented Programming (OOP)

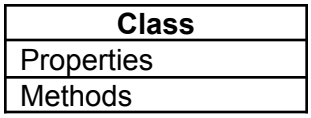
Object-oriented programming (OOP) is a form of programming that groups the data items and the subroutines that operate on the data items in one place so that only these subroutines can access the data items.

The data of an object are called **attributes** (or **properties**) and the subroutines acting on the attributes are called **methods**.

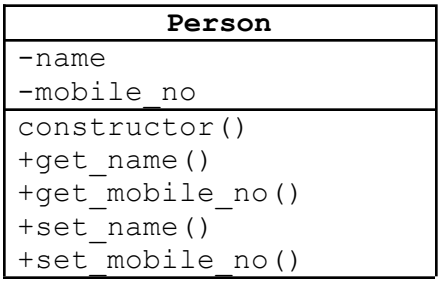
The 2 main building blocks of object-oriented programming are **classes** and **objects**.

A class is a blueprint or a template for an object that contains attributes or properties which describe the data, and methods which can access the attributes. When a class has been defined, it can be used to create one or more objects of this class type. Instantiation is the process of creating a new object. An object is an instance of a class. The objects will have the same attributes and methods as the class from which it was created.

A class can be represented using a class diagram. Each class is made up of 3 sections: the class name, properties (attributes) and methods.



The class diagram below shows an example of a `Person` class.



When we first set up an object of a particular class, we use a constructor. A **constructor** instantiates the object and assigns initial values to the attributes.

A `Person` class has 2 attributes: `name` and `mobile_no`. When we first create a `Person` object, we use a constructor method `__init()__` to instantiate the object and assign values to the attributes.

```
1  class Person:
2
3      # constructor
4      def __init__(self, name, mobile_no = ""):
5          self.name = name
6          self.mobile_no = mobile_no
```

The constructor method will automatically be called to initialise the newly created object.

The first input parameter for all instance methods must be `self`. The `self` parameter is referred to the current object of the class, i.e. the instance calling the method. To access any instance method or instance attribute in the class, you need to prefix it with `self`.

Note that the default of the parameter `mobile_no` can be set as an empty string as not everyone may have a mobile number.

Let's now create a new object: John with mobile number 91234567.

```
9  # main
10 p1 = Person("John", "91234567")
11 print(p1)
12
```

```
<__main__.Person object at 0x000001CA64C5F7B8>
```

Line 10 of the code above shows a new object `p1` is created and the object is stored at a particular address. To print out useful information of the object that is readable, we can implement the `__str__()` method.

```
1  class Person:
2
3      # constructor
4      def __init__(self, name, mobile_no = ""):
5          self.name = name
6          self.mobile_no = mobile_no
7
8      def __str__(self):
9          return "Name: {} \nMobile no: {}".format(self.name, self.mobile_no)
10
11 # main
12 p1 = Person("John", "91234567")
13 print(p1)
14
```

```
Name: John
Mobile no: 91234567
```

We can also write a display method to display the details of the data in the class.

```
    def display_details(self):
        print("Name: {} \nMobile no: {}".format(self.name, self. mobile_no))

# main
p1 = Person("John", "91234567")
p1.display_details()

p2 = Person("Mary")
p2.display_details()

```

```
Name: John
Mobile no: 91234567
Name: Mary
Mobile no:
```

Note that when you call an object's method, you do not need to give a value for the `self` parameter. Python provides it automatically.

1.1 Class attributes

Class Attributes are attributes which belong to the class instead of a particular object.

It can be accessed through either class or instance.

Class attributes can be used to keep a rolling value which is shared among all instances. For example, we would like to keep track of number of Customers by assigning each customer a unique serial number.

```
class Customer:

    current_serial = 0

    def __init__(self):
        #Either type(self).current_serial or
        #Customer.current_serial can be used to access the class variable current_serial
        type(self).current_serial += 1
        self.serial = Customer.current_serial

## Test
c1 = Customer()
c2 = Customer()
c3 = Customer()
print(c1.serial)
print(c2.serial)
print(c3.serial)
print(Customer.current_serial, c1.current_serial, c2.current_serial)
```

```
1
2
3
3 3 3
```

Modification to a class attribute can only be done with the notation `ClassName.AttributeName`. Otherwise, a new instance variable will be created.

1.2 Instance attribute vs Class attribute

Instance attributes belong to a particular instance. Modifying instance attribute of an instance does not affect other instances.

It is not necessary to declare an attribute for an object before using it either.

Note that assigning value to an non-existence attribute will create that attribute.

Class attributes are shared among all instances. They can be accessed not only through class but also through an instance.

```
1  class A:
2      cls_attr = "class attribute"
3
4  x = A()
5  print(x.cls_attr)
6
7  A.cls_attr = "my value"
8  print(x.cls_attr)
9  print(x.cls_attr is A.cls_attr)
```

```
class attribute
my value
True
```

In the above example, what if line 7 is changed to `x.cls_attr = "my value"` ?

2 Encapsulation

The idea behind OOP is that attributes can only be accessed through methods provided in the class. The direct path to the data is unavailable. Attributes are referred to as 'private'. The methods to access the data are made available to programmers, so these are 'public'. The feature of combining attributes and methods together in a single class is known as **encapsulation**. Encapsulation supports information hiding through the combining of private properties and public methods into a class, ensuring private properties are only accessed/altered by calls to public methods. It also supports implementation independence, which is the use of methods of a class without needing to know how it is implemented. Even if the underlying implementation were to change, it does not matter to the user.

In a class diagram, negative prefix denotes 'private' and positive prefix denotes 'public'. Hence, data should be tagged with negative signs to signify that they are private attributes and methods should be tagged with positive signs to signify that they are public methods.

Person
-name -mobile_no
+constructor() +get_name() +get_mobile_no() +set_mobile_no() +set_mobile_no()

When designing a class, we need to think about the attributes we want to store and the methods we need to access the data.

The methods used to get attributes of the object are usually referred to as **getters**.

Any attributes that might be updated after instantiation will need subroutines to update their values. These are referred to as **setters**. Some attributes get set only at instantiation. For attributes that are designed not to be changed, do not need setters. Hence, setters are provided only for attributes values that are to be changed.

```
1  class Person:
2
3      # constructor
4      def __init__(self, name, mobile_no = ""):
5          self.name = name
6          self.mobile_no = mobile_no
7
8      # getters
9      def get_name(self):
10         return self.name
11
12     def get_mobile_no(self):
13         return self.mobile_no
14
15     # setters
16     def set_name(self, name):
17         self.name = name
18
19     def set_mobile_no(self, mobile_no):
20         self.mobile_no = mobile_no
21
22     def __str__(self):
23         return "Name: {} \nMobile no: {}".format(self.name, self. mobile_no)
```

```
25 # main
26 p1 = Person("John", "91234567")
27 print(p1)
28
29 p2 = Person("Mary")
30 print(p2)
31
```

```
Name: John
Mobile no: 91234567
Name: Mary
Mobile no:
```

Back to the idea of private attributes, they should always be declared as 'Private'. One way of declaring an attribute as private is to put an underscore before an attribute name.

```
1 class Person:
2
3     # constructor
4     def __init__(self, name, mobile_no = ""):
5         self._name = name
6         self._mobile_no = mobile_no
```

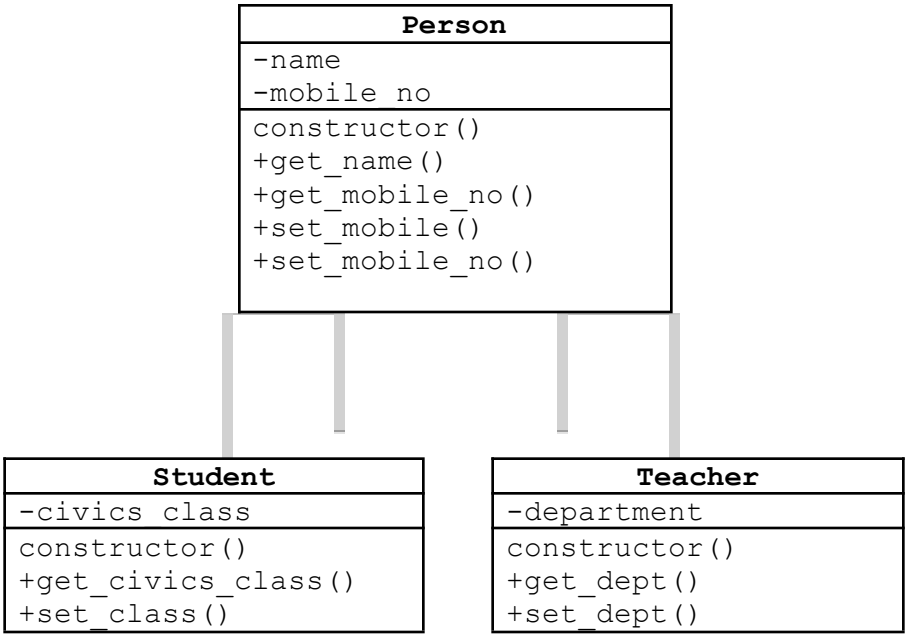
However, in Python, all the attributes are available to all users, even with an underscore prefix to the attribute names. While it does not prevent users from modifying the attributes, the underscore is to signify that the attributes are private, and it reminds the users that they would need to bear the consequences if their modifications to the attributes cause errors.

3 Inheritance

One advantage of object-oriented programming (OOP) is the ability to use **inheritance**. This allows us to define a base class (also called a superclass) and then create subclasses that are derived from it. These subclasses are automatically built upon the structure of the base class, meaning they come with the same attributes and methods already defined in the base class, reducing the need to duplicate code. In addition to inherited features, subclasses can also have their own unique attributes and methods. This promotes code reuse and makes programs easier to maintain and extend.

For example, in a school setting, there can be two types of persons, students and teachers. Both students and teachers have names and mobile numbers, however, students belong to classes while teachers belong to departments. Hence students and teachers will have common attributes of name and mobile number, and at the same time, they will have their own attributes and methods.

Below is an example of the class diagram showing superclass and subclasses.



Class diagrams with inheritance always have the superclass (base class) at the top, and the subclass(es) shown beneath. The direction of inheritance is always from subclass(es) to superclass.

The code below shows the base class `Person`.

```
class Person:

    # constructor
    def __init__(self, name, mobile_no = ""):
        self.name = name
        self.mobile_no = mobile_no

    # getters
    def get_name(self):
        return self.name

    def get_mobile_no(self):
        return self.mobile_no

    # setters
    def set_name(self, name):
        self.name = name

    def set_mobile_no(self, mobile_no):
        self.mobile_no = mobile_no

    def display_details(self):
        print("Name: {} \nMobile no: {}".format(self.name, self.mobile_no))
```

The code below shows how a subclass `Student` is declared in Python.

```
1  class Student(Person):
2
3      def __init__(self, name, mobile_no, c_class):
4          self.set_name(name)
5          self.set_mobile_no(mobile_no)
6          self.civics_class = c_class
7
8      def get_class(self):
9          return self.civics_class
10
11     def set_class(self, c_class):
12         self.civics_class = c_class
13
14 s1 = Student("Peter", "91111111", "22S57")
15 s1.display_details()
16
```

```
Name: Peter
Mobile no: 91111111
```

Line 1 of the code above indicates that the `Student` class is derived from the `Person` class and inherits all the attributes and behaviors (methods) defined in the `Person` class. Line 15 of the code above shows `Student` object `s1` calling the `display_details()` method inherited from its superclass. It can also have its own unique attributes (`civics_class`) and behaviors (`get_class()` and `set_class()`).

Inheriting attributes and methods in the base class promotes software reuse and avoid code duplication. To be able to use existing code, which is already tested, improves code reliability. Hence, debugging of software can be less tedious.

4 Polymorphism

All attributes and methods are inherited from the base class by the subclasses.

Polymorphism occurs when subclasses define methods with the same name as the methods in base class, but with different implementations. This allows the method in the derived class to override the one in the base class, so that each derived class can behave differently even when using the same method name.

Looking at the code snippets below, the constructor of the superclass is redefined in the subclasses.

```
class Student(Person):

    def __init__(self, name, mobile_no, c_class):
        self.name = name
        self.mobile_no = mobile_no
        self.civics_class = c_class

class Teacher(Person):

    def __init__(self, name, mobile_no, dept):
        self.name = name
        self.mobile_no = mobile_no
        self.dept = dept
```

When a method is redefined in the subclasses, the method in the subclasses has overridden the method in the super class. This is also known as method overriding.

To call a method in the superclass that has been overridden by the subclass, **super()** function can be used.

```
class Student(Person):

    def __init__(self, name, mobile_no, c_class):
        super().__init__(name, mobile_no)
        self.civics_class = c_class


class Teacher(Person):

    def __init__(self, name, mobile_no, dept):
        super().__init__(name, mobile_no)
        self.dept = dept
```

Referring to the code above, the constructors in the subclasses call the constructor of the superclass using the super() function and initializes the attributes in the superclass.

This helps to cut down on code duplication as the code from the parent class has been generalised and can be reused in all subclasses.
Method overriding is useful as we can redefine the same method to behave differently.

The `display_details()` method currently defined in the superclass only provide information on the attributes that are part of the superclass. This method can be redefined in the subclasses to include the additional attributes from the subclasses.

```
class Student(Person):

    def __init__(self, name, mobile_no, c_class):
        super().__init__(name, mobile_no)
        self.civics_class = c_class

    def get_class(self):
        return self.civics_class

    def set_class(self, c_class):
        self.civics_class = c_class

    def display_details(self):
        print("Name: {} \nMobile no: {}\nCivics class: {}".format(self.name, self.mobile_no, self.civics_class))


```

```
class Teacher(Person):

    def __init__(self, name, mobile_no, dept):
        Person.__init__(self, name, mobile_no)
        self.dept = dept

    def get_dept(self):
        return self.dept

    def set_dept(self, dept):
        self.dept = dept

    def display_details(self):
        print("Name: {} \nMobile no: {}\nDepartment: {}".format(self.name, self. mobile_no, self.dept))
```

The code above shows that the subclasses have overridden the method `display_details()` in the super class. The `display_details()` method in the subclasses now displays the new attributes of the respective subclasses.

```
s1 = Student("Peter", "91111111", "22S57")
s1.display_details()

t1 = Teacher("Mrs Yong", "92222222", "Computing")
t1.display_details()
```

```
Name: Peter
Mobile no: 91111111
Civics class: 22S57
Name: Mrs Yong
Mobile no: 92222222
Department: Computing
```

In summary, object-oriented programming (OOP) offers several advantages.

Encapsulation involves hiding information within objects by encapsulating data and methods within classes. This allows developers to control access to data, enhancing security and modularity. By preventing direct access to private attributes and allowing access only through public methods, data integrity is maintained, ensuring that modifications are done deliberately. Encapsulation also ensures that associated actions or behaviors are triggered, enabling the execution of specific tasks when certain conditions are met.

Inheritance promotes software reuse and implementation independence. It enables classes to inherit attributes and behaviors from other classes, establishing a hierarchy of classes where specialized classes can be created based on a common set of features. This allows developers to leverage existing code and extend it without rewriting definitions, leading to more efficient development. Furthermore, changes made within a class do not affect the overall system, ensuring that modifications are contained and localized.

Polymorphism enables code generalization by allowing objects of different classes to be treated uniformly, providing flexibility and extensibility in code design. These features enhance code organization, maintainability, and flexibility in building complex systems.

Overall, object-oriented programming and its key ideas empower developers to create well-organized, maintainable, and flexible systems by leveraging the principles of encapsulation, inheritance, and polymorphism.