Encapsulation in Python (OOP)

Encapsulation: Hiding Information

Encapsulation is a mechanism that restricts direct access to objects' data and methods. But at the same time, it facilitates operation on that data (objects' methods).

> "Encapsulation can be used to hide data members and members function. Under this definition, encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition." — Wikipedia

All internal representation of an object is hidden from the outside. Only the object can interact with its internal data.

First, we need to understand how `public` and `non-public` instance variables and methods work.

> Public Instance Variables

For a Python class, we can initialize a `public instance variable` within our constructor method. Let's see this:

Within the constructor method:

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name
```

Here we apply the `first_name` value as an argument to the `public instance variable`.

```
tk = Person('TK')
print(tk.first_name) # => TK
```

Within the class:

```
class Person:
   first_name = 'TK'
```

Here, we do not need to apply the first_name as an argument, and all instance objects will have a class attribute initialized with TK.

```
tk = Person()
print(tk.first_name) # => TK
```

Cool. We have now learned that we can use public instance variables and class attributes. Another interesting thing about the public part is that we can manage the variable value. What do I mean by that? Our object can manage its variable value: Get and Set variable values.

Keeping the Person class in mind, we want to set another value to its first_name variable:

```
tk = Person('TK')
tk.first_name = 'Kaio'
print(tk.first_name) # => Kaio
```

Here we go. We just set another value (kaio) to the first_name instance variable and it updated the value. Simple as that. Since it's a public variable, we can do that.

> Non-public Instance Variable

We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work). — PEP 8

As the public instance variable , we can define the non-public instance variable both within the constructor method or within the class. The syntax difference is: for non-public instance variables , use an underscore (_) before the variable name.

"'Private' instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member)" — Python Software Foundation

Here's an example:

```
class Person:
  def __init__(self, first_name, email):
    self.first_name = first_name
    self._email = email
```

Did you see the email variable? This is how we define a non-public variable :

```
tk = Person('TK', 'tk@mail.com')
print(tk._email) # tk@mail.com
```

We can access and update it. *Non-public variables* are just a convention and should be treated as a non-public part of the API.

So we use a method that allows us to do it inside our class definition. Let's implement two methods (email and update_email) to understand it:

```
class Person:
  def __init__(self, first_name, email):
    self.first_name = first_name
    self._email = email

  def update_email(self, new_email):
    self._email = new_email

  def email(self):
    return self._email
```

Now we can update and access non-public variables using those methods. Let's see:

```
tk = Person('TK', 'tk@mail.com')
print(tk.email()) # => tk@mail.com
tk._email = 'new_tk@mail.com'
print(tk.email()) # => tk@mail.com
tk.update_email('new_tk@mail.com')
print(tk.email()) # => new_tk@mail.com
```

1. We initiated a new object with first_name TK and email tk@mail.com
2. Printed the email by accessing the non-public variable with a method
3. Tried to set a new email out of our class
4. We need to treat non-public variable as non-public part of the API
5. Updated the non-public variable with our instance method
6. Success! We can update it inside our class with the helper method

> Public Method

With public methods, we can also use them out of our class:

```
class Person:
  def __init__(self, first_name, age):
    self.first_name = first_name
    self._age = age

  def show_age(self):
    return self._age
```

Let's test it:

```
tk = Person('TK', 25)
print(tk.show_age()) # => 25
```

Great — we can use it without any problem.

> Non-public Method

But with non-public methods we aren't able to do it. Let's implement the same Person class, but now with a show_age non-public method using an underscore (_).

```
class Person:
  def __init__(self, first_name, age):
    self.first_name = first_name
    self._age = age

  def _show_age(self):
    return self._age
```

And now, we'll try to call this non-public method with our object:

```
tk = Person('TK', 25)
print(tk._show_age()) # => 25
```

We can access and update it. *Non-public methods* are just a convention and should be treated as a non-public part of the API.

Here's an example for how we can use it:

```
class Person:
  def __init__(self, first_name, age):
    self.first_name = first_name
    self._age = age

  def show_age(self):
    return self._get_age()

  def _get_age(self):
    return self._age

tk = Person('TK', 25)
print(tk.show_age()) # => 25
```

Here we have a _get_age non-public method and a show_age public method. The show_age can be used by our object (out of our class) and the _get_age only

used inside our class definition (inside `show_age` method). But again: as a matter of convention.

Encapsulation Summary

With encapsulation we can ensure that the internal representation of the object is hidden from the outside.