Encapsulation in Python

Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.

- Defining a class in object-oriented programming (OOP) ends up with some instance and class attributes.
- Attributes are just variables that you can access through the instance, the class, or both.

Attributes hold the internal state of objects. To access and mutate them, we have at least two ways to access and mutate attributes. You can either:

1. Access and mutate the attribute **directly**
2. Use **methods** to access and mutate the attribute

Getter and setter methods are quite popular in many object-oriented programming languages:

- **Getter:** A method that allows you to *access* an attribute in a given class
- **Setter:** A method that allows you to *set* or *mutate* the value of an attribute in a class

In OOP, the getter and setter pattern suggests that public attributes should be used only when you’re sure that no one will ever need to attach behavior to them. If an attribute is likely to change its internal implementation, then you should use getter and setter methods.

Implementing the getter and setter pattern requires:

1. **Making your attributes non-public**
2. **Writing getter and setter methods for each attribute**

Example:

```
class Employee:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    def get_name(self):
        return self._name

    def set_name(self, value):
        self._name = value

    def get_age(self):
        return self._age

    def set_age(self, value):
        self._age = value
```

```
# instance
emp1=Employee("name_test",25)
# try to get the name
emp1.name
# try to get _name
emp1._name
```

Conclusion:

Python doesn't have the notion of access modifiers, such as `private`, `protected`, and `public`, to restrict access to attributes and methods in a class. In Python, the distinction is between **public** and **non-public** class members.

Using Properties

Properties represent an intermediate functionality between a plain attribute (or field) and a method. In other words, they allow you to create methods that behave like attributes.

With properties, you can change how you compute the target attribute whenever you need to do so.

> 💡 **Note:** It's common to refer to `property()` as a built-in function. However, `property` is a class designed to work as a function rather than as a regular class.

Note: full signature of `property()`:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Argument	Description
`fget`	Function that returns the value of the managed attribute
`fset`	Function that allows you to set the value of the managed attribute
`fdel`	Function to define how the managed attribute handles deletion
`doc`	String representing the property's docstring

Example:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """The radius property."""
        print("Get radius")
        return self._radius

    @radius.setter
    def radius(self, value):
        print("Set radius")
        self._radius = value
```

```
    @radius.deleter
    def radius(self):
        print("Delete radius")
        del self._radius
```

```
circle=Circle(25)
dir(Circle.radius)
circle.radius = 100.0
circle.radius
del circle.radius
```

Note

- The `@property` decorator must decorate the **getter method**.
- The docstring must go in the **getter method**.
- The **setter and deleter methods** must be decorated with the name of the getter method plus `.setter` and `.deleter`, respectively.