

Object-Oriented Programming

Access modifiers and Properties

KHANG NGUYỄN CÔNG GIÁ

February 2026

Chapter 1

Access Modifiers

1.1 Encapsulation

The process of wrapping up variables and methods into a single entity is known as **Encapsulation**. It is one of the underlying concepts in object-oriented programming (OOP). Encapsulation in Python acts as a protective shield that puts restrictions on accessing variables and methods directly, and can prevent accidental or unauthorized modification of data. Encapsulation also makes objects into more autonomous, independently functioning pieces.

Moreover, encapsulation is not only about **hiding details** but also about **controlling access** and **safeguarding** an object's internal state.

Encapsulation is vital in Python programming because it helps you write well-organized, readable code. It ensures that the inner workings of your classes remain hidden from the outside world, making your application more secure and easier to maintain.

1.2 Access modifiers in OOP languages

Most OOP languages like C++ and Java use access modifiers to control access to class members (variables and methods). These modifiers typically include:

- **Public** access modifiers: Accessible from anywhere.
- **Protected** access modifiers: Accessible within the class and by subclasses.

- **Private** access modifiers: Accessible only within the class.

Python does not have built-in access modifiers like C++ or Java. All class variables and methods are **public** by default. However, Python follows a convention to imitate access control by prefixing variable/method names with **underscores**.

1.3 Access Modifiers in Python

1.3.1 Public Access Modifier

- Members (variables or methods) declared as public can be accessed from **anywhere** in the program.
- By **default**, all members are public in Python.

1.3.2 Protected Access Modifier

- A member is considered protected if its name starts with a **single underscore** (_).
- Convention only: It suggests that the member **should not** be accessed outside the class except by **subclasses**.
- Still, Python allows direct access if explicitly called.

1.3.3 Private Access Modifier

- A member is private if its name starts with **double underscores** (_).
- Python does not enforce strict privacy — instead, it uses **Name Mangling**.
- The interpreter renames `__var` → `_ClassName__var` internally.

1.4 Example

```

1  class Person:
2      def __init__(self, name, age, height):
3          self.name      = name    # public
4          self._age     = age     # protected
5          self.__height = height  # private
6
7  Khang = Person("Khang", 17, 170)

```

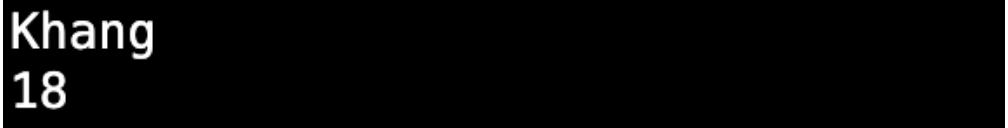
```

8     print(Khang.name)           # public: can be accessed
9     print(Khang._age)          # protected: can be accessed but not
10    →   advised
10    # print(p1.__height) # private: will give AttributeError

```

1.5 Outputs

Run the program and we will have this output:



```

Khang
18

```

Figure 1.1: Outputs of accessing Public and Protected attributes of 'Person' class

If we try to access the Private attribute, the program will return an error:

```

Traceback (most recent call last):
  File "/home/repl660/main.py", line 12, in <module>
    print(Khang.__height) # private: will give AttributeError
                           ^^^^^^^^^^
AttributeError: 'Person' object has no attribute '__height'

```

Figure 1.2: Error raised when user try to access the Private attributes

Note: In Python, access modifiers are not strictly enforced like in some other programming languages. Instead of blocking access completely, Python relies on conventions and developer responsibility. To better control how data is accessed and modified, Python provides **properties**. Properties allow us to manage how an attribute is read, changed, or deleted, while still keeping the same simple syntax when using the object. In this way, properties help solve the limitations of traditional access modifiers and provide a cleaner and more flexible way to protect and control data in object-oriented programming.

Chapter 2

Properties

The **property protocol** allows us to route a specific attribute's **get**, **set**, and **delete** operations to functions or methods we provide, enabling us to insert code to be run automatically on **attribute access**, **intercept attribute deletions**, and **provide documentation** for the attributes if desired.

A property manages a single, specific attribute; although it can't catch all attribute accesses generically, it allows us to control both fetch and assignment accesses and enables us to change an attribute from simple data to a computation freely, without breaking existing code. Properties are strongly related to descriptors; in fact, they are essentially a restricted form of them.

2.1 The basic

A property is created by assigning the result of a built-in function to a class attribute:

```
1 attribute = property(fget, fset, fdel, doc)
```

None of this built-in's arguments are required, and all default to **None** if not passed. For the first three, this **None** means that the corresponding operation is not supported, and attempting it will raise an `AttributeError` exception automatically.

When these arguments are used, we pass `fget` a function for intercepting attribute fetches, `fset` a function for assignments, and `fdel` a function for attribute deletions. Technically, all three of these arguments accept any callable, including a class's method, having a first argument to receive the instance being qualified. When later invoked, the `fget` function returns the computed attribute value, `fset` and `fdel`

return nothing (really, None), and all three may raise exceptions to reject access requests.

The doc argument receives a documentation string for the attribute, if desired; otherwise, the property copies the docstring of the fget function, which as usual defaults to None.

This built-in property call returns a property object, which we assign to the name of the attribute to be managed in the class scope, where it will be inherited by every instance.

2.2 Coding Properties with Decorators

A function decorator is a kind of runtime declaration about the function whose definition follows. The decorator is coded on a line just before the def statement that defines a function or method, and it consists of the @ symbol followed by a reference to a metaprogramming—a function (or other callable object) that manages another function. In terms of code, function decorators automatically map the following syntax:

```
1 @decorator # Decorate function
2 def F(arg):
3     ...
4     F(99) # Call function
```

into this equivalent form, where decorator is a one-argument callable object that returns a callable object with the same number of arguments as F (in not F itself):

```
1 def F(arg):
2     ...
3     F = decorator(F) # Rebind function name to decorator result
4     F(99) # Essentially calls decorator(F)(99)
```

Because of this mapping, it turns out that the property built-in can serve as a decorator, to define a function that will run automatically when an attribute is fetched:

```
1 class Person:
2     @property
3         def name(self): ... # Rebinds: name = property(name)
```

When run, the decorated method is automatically passed to the first argument of the property built-in. This is really just alternative syntax for creating a property and rebinding the attribute name manually, but may be seen as more explicit in this role:

```
1 class Person:  
2     def name(self): ...  
3     name = property(name)
```

2.3 Example

Now I will show you the clear example of how property can help us in access modifier with the same class Person as before:

```
1 class Person:  
2     def __init__(self, name, age, height):  
3         self._name = name #protected  
4         self.__age=age #private  
5         self.height=height #public  
6  
7     #Properties to access protected attribute:  
8     @property  
9     def name(self): # name = property(name)  
10        print('Get name: ')  
11        return self._name  
12  
13     @name.setter  
14     def name(self, value): # name = name.setter(name)  
15        print(f'Change name from {self._name} to: {value}')  
16        self._name = value  
17  
18     @name.deleter  
19     def name(self): # name = name.deleter(name)  
20        print('Removed name')  
21        del self._name  
22  
23     #Properties to access private attribute:  
24     @property  
25     def age(self): # get age function  
26        print('Get age: ')  
27        return self.__age
```

```

29     @age.setter
30     def age(self, value): # set age function
31         print(f'Change age from {self.__age} to: {value}')
32         self.__age = value
33
34     @age.deleter
35     def age(self): # delete age function
36         print('Removed age')
37         del self.__age
38
39
40 Khang = Person('Nguyen Khang',18,170) # Khang has a managed attribute
41 print(Khang.name) # Runs name getter
42 Khang.name = 'Nguyen Cong Gia Khang' # Runs name setter
43 print(Khang.name)
44 del Khang.name # Runs name deleter
45 print(Khang.age) # Runs age getter
46 Khang.age = 19 #Runs age setter
47 print(Khang.age)
48 del Khang.age # Runs age deleter
49 print(Khang.height) # Get public height value
50 Khang.height=180 # Change public height value
51 print(Khang.height)
52 del Khang.height # Delete public height value

```

2.4 Outputs

```
Get name:  
Nguyen Khang  
Change name from Nguyen Khang to: Nguyen Cong Gia Khang  
Get name:  
Nguyen Cong Gia Khang  
Removed name  
Get age:  
18  
Change age from 18 to: 19  
Get age:  
19  
Removed age  
170  
180
```

Figure 2.1: Code results for testing the properties of the 'name' and 'age' attribute.

So this output shows that with the help of **Property**, we can access the protected and private attributes and perform actions like get, set and delete.

Chapter 3

References

- [1] Lutz, M. (2013). Learning Python,Fifth Edition.
- [2] <https://www.educative.io/answers/what-are-public-protected-private-access-modifiers-in-python>
- [3] <https://www.geeksforgeeks.org/python/access-modifiers-in-python-public-private-and-protected/>