Python Private Attributes

Summary: in this tutorial, you'll learn about encapsulation and how to use private attributes to accomplish encapsulation in Python.

Introduction to encapsulation in Python

Encapsulation is one of the four fundamental concepts in object-oriented programming including abstraction, encapsulation, inheritance, and polymorphism.

Encapsulation is the packing of data and functions (https://www.pythontutorial.net/python-basics/python-functions/) that work on that data within a single object. By doing so, you can hide the internal state of the object from the outside. This is known as **information hiding**.

A class (https://www.pythontutorial.net/python-oop/python-class/) is an example of encapsulation. A class bundles data and methods into a single unit. And a class provides the access to its attributes via methods.

The idea of information hiding is that if you have an attribute that isn't visible to the outside, you can control the access to its value to make sure your object is always has a valid state.

Let's take a look at an example to better understand the encapsulation concept.

Python encapsulation example

The following defines the `Counter` class:

```
class Counter:
    def __init__(self):
        self.current = 0

    def increment(self):
        self.current += 1

    def value(self):
        return self.current

    def reset(self):
        self.current = 0
```

The `Counter` class has one attribute called `current` which defaults to zero. And it has three methods:

- `increment()` increases the value of the `current` attribute by one.
- `value()` returns the current value of the `current` attribute
- `reset()` sets the value of the `current` attribute to zero.

The following creates a new instance of the `Counter` class and calls the `increment()` method three times before showing the current value of the counter to the screen:

```
counter = Counter()


counter.increment()
counter.increment()
counter.increment()

print(counter.value())
```

Output:

```
3
```

It works perfectly fine but has one issue.

From the outside of the `Counter` class, you still can access the current attribute and change it to whatever you want. For example:

```
counter = Counter()

counter.increment()
counter.increment()
counter.current = -999

print(counter.value())
```

Output:

```
-999
```

In this example, we create an instance of the `Counter` class, call the `increment()` method twice and set the value of the current attribute to an invalid value `-999` .

So how do you prevent the `current` attribute from modifying outside of the `Counter` class?

That’s why private attributes come into play.

Private attributes

Private attributes can be only accessible from the methods of the class. In other words, they cannot be accessible from outside of the class.

Python doesn’t have a concept of private attributes. In other words, all attributes are accessible from the outside of a class.

By convention, you can define a private attribute by prefixing a single underscore (_):

```
_attribute
```

This means that the _attribute should not be manipulated and may have a breaking change in the future.

The following redefines the `Counter` class with the `current` as a private attribute by convention:

```
class Counter:
    def __init__(self):
        self._current = 0

    def increment(self):
        self._current += 1

    def value(self):
        return self._current

    def reset(self):
        self._current = 0
```

Name mangling with double underscores

If you prefix an attribute name with double underscores (`__`) like this:

```
__attribute
```

Python will automatically change the name of the `__attribute` to:

```
_class__attribute
```

This is called the name mangling in Python.

By doing this, you cannot access the __attribute directly from the outside of a class like:

```
instance.__attribute
```

However, you still can access it using the _class__attribute name:

```
instance._class__attribute
```

The following example redefines the Counter class with the __current attribute:

```
class Counter:
    def __init__(self):
        self.__current = 0

    def increment(self):
        self.__current += 1

    def value(self):
        return self.__current

    def reset(self):
        self.__current = 0
```

Now, if you attempt to access __current attribute, you'll get an error:

```
counter = Counter()
print(counter.__current)
```

Output:

```
AttributeError: 'Counter' object has no attribute '__current'
```

However, you can access the __current attribute as _Counter___current like this:

```
counter = Counter()
print(counter._Counter__current)
```

Summary

- Encapsulation is the packing of data and methods into a class so that you can hide the information and restrict access from outside.
- Prefix an attribute with a single underscore (`_`) to make it private by convention.
- Prefix an attribute with double underscores (`__`) to use the name mangling.