



جامعة الانبار
كلية علوم الحاسوب وتكنولوجيا المعلومات
قسم أنظمة شبكات الحاسوب

برمجة كيانية OOP
المرحلة الثانية
الفصل الدراسي الأول والثاني

مدرس المادة
م.د. سميه عبدالله حمد

Object Oriented Programming (OOP)

Private Variables and Methods

Private Variables

When instance variables of the parent class don't need to be inherited by the child class, they can be made unavailable to the child class by adding double underscores (__) before the variable name. This appends `__classname` before the variable name. So, when we try to access it like other instance variables of the class it gives an "Attribute error".

```
class ProgramLanguage:
    def __init__(self, name):
        self.name = name
        self.__private = 'Private variable'
class Python(ProgramLanguage):
    pass
y = Python("Python")
print(y.name)
print(y.__private)
```

The error would be:

AttributeError: 'Python' object has no attribute '__private'

`__private` in `parent` class `ProgramLanguage` is now `__ProgramLanguage__private` and can't be accessed using `.__private`. Same is true for class methods.

Note: Employing a double underscore prefix only makes the method or variable inaccessible using the originally declared name. These can still be accessed, like `y._ProgramLanguage__private` in the above example. The use of this syntax is an indication of how the variable or method should be treated.

Object Oriented Programming (OOP)

Private Methods in Python

In Python, you can define a private method by prefixing the method name with a single underscore. Differently from other programming languages making a method private in Python doesn't prevent you from accessing it from outside your class. It's simply a convention to tell other developers that the method is for "internal use only" for your class.

A private method is a method that should only be called inside the Python class where it is defined. To indicate a Python private method prefix its name with a single underscore.

Let's see how you can define a private method in Python and how it differs from a public method.

```
class Person:

    def __init__(self, name):

        self.name = name

    def run(self):

        print("{} is running".format(self.name))

jack = Person("Jack")

jack.run()
```

When we create an instance and execute the run() method we get back the expected message:

Output:

Jack is running

Object Oriented Programming (OOP)

Now let's say we want to add a `warmup()` method that the `run` method calls internally. At the same time, the `warmup()` method shouldn't be callable outside of the class.

In theory, we can achieve this by adding **one underscore** before the name of the method:

```
def run(self):  
    self._warmup()  
    print("{} is running".format(self.name))  
  
def _warmup(self):  
    print("{} is warming up".format(self.name))
```

As you can see we have defined the `_warmup()` method and then we call the private method inside the `run()` method.

Nothing changes in the way you call `run()` on the instance we have created before:

```
jack = Person("Jack")  
  
jack.run()
```

Output:

```
Jack is warming up  
Jack is running
```

Now, let's see what happens if we try to call the `_warmup()` method directly on the `Person` instance.

```
jack._warmup()
```

Output:

```
Jack is warming up
```

Object Oriented Programming (OOP)

Using a single underscore to indicate the name of a Python private method in a class is just a naming convention between developers and it's not enforced by the Python interpreter.

A private method defined in a Python class should not be called on an instance of that class. It should only be called inside the class itself.

In Python, it's also possible to prefix the name of a method with a double underscore instead of a single underscore.

Update the method `_warmup()` from the previous example and add another underscore at the beginning of the method name: `__warmup()`.

```
def run(self):  
    self.__warmup()  
    print("{} is running".format(self.name))  
  
def __warmup(self):  
    print("{} is warming up".format(self.name))
```

The `run()` method behaves in the same way when called on the instance:

```
jack = Person("Jack")  
  
jack.run()
```

Output:

Jack is warming up

Jack is running

Object Oriented Programming (OOP)

And what happens if we call the method `__warmup()` on the Person instance?

```
jack.__warmup()
```

Output:

Traceback (most recent call last):

File "private.py", line 45, in <module>

```
jack.__warmup()
```

AttributeError: 'Person' object has no attribute '__warmup'

The Python interpreter throws an exception and tells us that this Person object has no attribute `__warmup`.

This error message could be misleading considering that this method is present in the class but the Python interpreter is “hiding” it by using something called **name mangling**.

The purpose of name mangling is to avoid collisions with method names when inheriting a class.

we have seen what happens when you prefix method names with two underscores. But, is the Python interpreter hiding these methods completely?

To answer this question we will use the `dir()` function to see attributes and methods available in our Person instance.

```
print(dir(jack))
```

Output:

```
['_Person__warmup', '__class__', '__delattr__', '__dict__',  
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',  
 '__getattr__', '__gt__', '__hash__', '__init__',  
 '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',  
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',  
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',  
 '__weakref__', 'name', 'run']
```

Object Oriented Programming (OOP)

Interestingly, we don't see `__warmup` in the list of available methods but we see `_Person__warmup`.

Let's try to call it on the instance:

```
jack = Person("Jack")  
  
jack._Person__warmup()
```

Output:

Jack is warming up

So it looks like our name mangled method is not completely hidden considering that we can access it by adding an underscore and the class name before the method name.

`_{class-name}__{name-mangled-method}`

In Python, you can access a method whose name starts with double underscores (and doesn't end with underscores) from an instance of a class. You can do that by adding an underscore and the name of the class before the name of the method. This is called **name mangling**.

Name Mangling and Inheritance in Python

Define a class called Runner that inherits the base class Person.

```
class Runner(Person):  
  
    def __init__(self, name, fitness_level):  
  
        super().__init__(name)  
  
        self.fitness_level = fitness_level
```

Object Oriented Programming (OOP)

Then execute the public method `run()` of the parent class on a `Runner` instance.

```
kate = Runner("Kate", "high")
```

```
kate.run()
```

Output:

```
Kate is warming up
```

```
Kate is running
```

The child class has inherited the public method `run()`. And what happens if we try to call the name mangled method?

```
kate.__warmup()
```

Output:

```
Traceback (most recent call last):
```

```
File "private.py", line 19, in <module>
```

```
kate.__warmup()
```

```
AttributeError: 'Runner' object has no attribute '__warmup'
```

We get the expected error due to name mangling. **Notice that** we can still call it by using the name of the parent class as we have seen in the previous section:

```
kate._Person__warmup()
```

Output:

```
Kate is warming up
```


Object Oriented Programming (OOP)

Below you can see the output of the dir() function for the child class.

```
['_Person__warmup', '__class__', '__delattr__', '__dict__',  
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',  
 '__getattr__', '__gt__', '__hash__', '__init__',  
 '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',  
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',  
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',  
 '__weakref__', 'fitness_level', 'name', 'run']
```

Defining a Name Mangled Method in a Child Class

What happens if we define the same name mangled method in our child class?

Override the public method run() and the “hidden” method __warmup() in the Runner class.

```
class Runner(Person):  
  
    def __init__(self, name, fitness_level):  
  
        super().__init__(name)  
  
        self.fitness_level = fitness_level  
  
    def run(self):  
  
        self.__warmup()  
  
        print("{} has started a race".format(self.name))  
  
    def __warmup(self):  
  
        print("{} is warming up before a race".format(self.name))  
  
kate = Runner("Kate", "high")  
  
kate.run()
```

Output:

```
Kate is warming up before a race  
Kate has started a race
```

Object Oriented Programming (OOP)

So, both methods in the child class are executed.

One thing I'm curious about it's how the Python interpreter represents the new name mangled method in the child object considering that for the parent object it was using an underscore followed by the class name.

```
print(dir(kate))
```

Output:

```
['_Person__warmup', '_Runner__warmup', '__class__',  
 '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',  
 '__format__', '__ge__', '__getattr__', '__gt__', '__hash__',  
 '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',  
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',  
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',  
 '__weakref__', 'fitness_level', 'name', 'run']
```

You can see that the child object has now two mangled methods:

- `_Person__warmup`
- `_Runner__warmup`

This shows how name mangling prevents collisions with method names when you inherit a class.

```
class Person:  
  
    def __init__(self, name):  
  
        self.name = name  
  
    def run(self):  
  
        self.__warmup()  
  
        print("{} is running".format(self.name))  
  
    def __warmup(self):  
  
        print("{} is warming up".format(self.name))
```

Object Oriented Programming (OOP)

```
class Runner(Person):  
  
    def __init__(self, name, fitness_level):  
  
        super().__init__(name)  
  
        self.fitness_level = fitness_level  
  
    def run(self):  
  
        self.__warmup()  
  
        print("{} has started a race".format(self.name))  
  
    def __warmup(self):  
  
        print("{} is warming up before a race".format (self.name))  
  
kate = Runner("Kate", "high")  
  
kate.run()
```

Output:

Kate is warming up before a race

Kate has started a race