PUBLIC, - PROTECTED-, AND PRIVATE ATTRIBUTES

Who doesn't know those trigger-happy farmers from films. Shooting as soon as somebody enters their property. This "somebody" has of course neglected the "no trespassing" sign, indicating that the land is private property. Maybe he hasn't seen the sign, maybe the sign is hard to be seen? Imagine a jogger, running the same course five times a week for more than a year, but than he receives a $50 fine for trespassing in the Winchester Fells. Trespassing is a criminal offence in Massachusetts. He was innocent anyway, because the signage was inadequate in the area**.

Even though no trespassing signs and strict laws do protect the private property, some surround their property with fences to keep off unwanted "visitors". Should the fence keep the dog in the yard or the burglar on the street? Choose your fence: Wood panel fencing, post-and-rail fencing, chain-link fencing with or without barbed wire and so on.

We have a similar situation in the design of object-oriented programming languages. The first decision to take is how to protect the data which should be private. The second decision is what to do if trespassing, i.e. accessing or changing private data, occurs. Of course, the private data may be protected in a way that it can't be accessed under any circumstances. This is hardly possible in practice, as we know from the old saying "Where there's a will, there's a way"!

Some owners allow a restricted access to their property. Joggers or hikers may find signs like "Enter at your own risk". A third kind of property might be public property like streets or parks, where it is perfectly legal to be.

We have the same classification again in object-oriented programming:

- Private attributes should only be used by the owner, i.e. inside of the class definition itself.
- Protected (restricted) Attributes may be used, but at your own risk. Essentially, they should only be used under certain conditions.
- Public Attributes can and should be freely used.

Python uses a special naming scheme for attributes to control the accessibility of the attributes. So far, we have used attribute names, which can be freely used inside or outside of a class definition, as we have seen. This corresponds to public attributes of course. There are two ways to restrict the access to class attributes:

- First, we can prefix an attribute name with a leading underscore "_". This marks the attribute as protected. It tells users of the class not to use this attribute unless, they write a subclass. We will learn about inheritance and subclassing in the next chapter of our tutorial.
- Second, we can prefix an attribute name with two leading underscores "__". The attribute is now inaccessible and invisible from outside. It's neither possible to read nor write to those attributes except inside the class definition itself*.

To summarize the attribute types:

Naming	Type	Meaning
name	Public	These attributes can be freely used inside or outside a class definition.
_name	Protected	Protected attributes should not be used outside the class definition, unless inside a subclass definition.
__name	Private	This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside the class definition itself.

We want to demonstrate the behaviour of these attribute types with an example class:

```
class A():

    def __init__(self):
        self.__priv = "I am private"
        self._prot = "I am protected"
        self.pub = "I am public"
```

We store this class (attribute_tests.py) and test its behaviour in the following interactive Python shell:

```
from attribute_tests import A
x = A()
x.pub
```

Output::

```
'I am public'
```

```
x.pub = x.pub + " and my value can be changed"
x.pub
```

Output::

```
'I am public and my value can be changed'
```

```
x._prot
```

Output::

```
'I am protected'
```

```
x.__priv
```

```
-----------------------------------------------------------------------
----------------
AttributeError                            Traceback (most r
ecent call last)
<ipython-input-6-f75b36b98afa> in <module>
----> 1 x.__priv

AttributeError: 'A' object has no attribute '__priv'
```

The error message is very interesting. One might have expected a message like " `__priv` is private". We get the message "AttributeError: 'A' object has no attribute `__priv` instead, which looks like a "lie". There is such an attribute, but we are told that there isn't. This is perfect information hiding. Telling a user that an attribute name is private, means that we make some information visible, i.e. the existence or non-existence of a private variable.

Our next task is rewriting our Robot class. Though we have Getter and Setter methods for the name and the build_year, we can access the attributes directly as well, because we have defined them as public attributes. Data Encapsulation means, that we should only be able to access private attributes via getters and setters.

We have to replace each occurrence of self.name and self.build_year by self. `__name` and `self.__build_year` .

The listing of our revised class:

```
class Robot:

    def __init__(self, name=None, build_year=2000):
        self.__name = name
        self.__build_year = build_year

    def say_hi(self):
        if self.__name:
            print("Hi, I am " + self.__name)
        else:
            print("Hi, I am a robot without a name")

    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_build_year(self, by):
        self.__build_year = by

    def get_build_year(self):
        return self.__build_year

    def __repr__(self):
        return "Robot('" + self.__name + "', " +  str(self.__build_
year) +  ")"

    def __str__(self):
        return "Name: " + self.__name + ", Build Year: " +  str(sel
f.__build_year)


if __name__ == "__main__":
    x = Robot("Marvin", 1979)
    y = Robot("Caliban", 1943)
    for rob in [x, y]:
        rob.say_hi()
        if rob.get_name() == "Caliban":
            rob.set_build_year(1993)
        print("I was built in the year " + str(rob.get_build_year
()) + "!")
```

```
Hi, I am Marvin
I was built in the year 1979!
Hi, I am Caliban
I was built in the year 1993!
```

Every private attribute of our class has a getter and a setter. There are IDEs for object-oriented programming languages, who automatically provide getters and setters for every private attribute as soon as an attribute is created.

This may look like the following class:

```
class A():

    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def GetX(self):
        return self.__x

    def GetY(self):
        return self.__y

    def SetX(self, x):
        self.__x = x

    def SetY(self, y):
        self.__y = y
```

There are at least two good reasons against such an approach. First of all not every private attribute needs to be accessed from outside. Second, we will create non-pythonic code this way, as you will learn soon.

DESTRUCTOR

What we said about constructors holds true for destructors as well. There is no "real" destructor, but something similar, i.e. the method `__del__` . It is called when the instance is about to be destroyed and if there is no other reference to this instance. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

The following script is an example with `__init__` and `__del__` :

```
class Robot():

    def __init__(self, name):
        print(name + " has been created!")

    def __del__(self):
        print ("Robot has been destroyed")


if __name__ == "__main__":
    x = Robot("Tik-Tok")
    y = Robot("Jenkins")
    z = x
    print("Deleting x")
    del x
    print("Deleting z")
    del z
    del y
```

```
Tik-Tok has been created!
Jenkins has been created!
Deleting x
Deleting z
Robot has been destroyed
Robot has been destroyed
```

The usage of the `__del__` method is very problematic. If we change the previous code to personalize the deletion of a robot, we create an error:

```
class Robot():

    def __init__(self, name):
        print(name + " has been created!")

    def __del__(self):
        print (self.name + " says bye-bye!")


if __name__ == "__main__":
    x = Robot("Tik-Tok")
    y = Robot("Jenkins")
    z = x
    print("Deleting x")
    del x
    print("Deleting z")
    del z
    del y
```

```
Tik-Tok has been created!
Jenkins has been created!
Deleting x
Deleting z

Exception ignored in: <function Robot.__del__ at 0x0000024A
13CDC8B8>
Traceback (most recent call last):
  File "<ipython-input-18-7dedd1b7f17b>", line 7, in __del_
_
AttributeError: 'Robot' object has no attribute 'name'
Exception ignored in: <function Robot.__del__ at 0x0000024A
13CDC8B8>
Traceback (most recent call last):
  File "<ipython-input-18-7dedd1b7f17b>", line 7, in __del_
_
AttributeError: 'Robot' object has no attribute 'name'
```

We are accessing an attribute which doesn't exist anymore. We will learn later, why this is the case.

Footnotes: + The picture on the right side is taken in the Library of the Court of Appeal for Ontario, located downtown Toronto in historic Osgoode Hall

++ "Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.) Every object has an identity, a type and a value." (excerpt from the official Python Language Reference)

+++ "attribute" stems from the Latin verb "attribuere" which means "to associate with"

++++ http://www.wickedlocal.com/x937072506/tJogger-ticketed-for-trespassing-in-the-Winchester-Fells-kicks-back

+++++ There is a way to access a private attribute directly. In our example, we can do it like this: x._Robot__build_year You shouldn't do this under any circumstances!