OOP: Properties

Instance variables

In general, a class can be equipped with two different kinds of data to form a class's properties. You already saw one of them when we were looking at stacks.

This kind of class property exists when and only when it is explicitly created and added to an object. As you already know, this can be done during the object's initialization, performed by the constructor.

Moreover, it can be done in any moment of the object's life. Furthermore, any existing property can be removed at any time.

Such an approach has some important consequences:

- different objects of the same class **may possess different sets of properties**;
- there must be a way to **safely check if a specific object owns the property** you want to utilize (unless you want to provoke an exception - it's always worth considering)
- each object **carries its own set of properties -** they don't interfere with one another in any way.

Such variables (properties) are called **instance variables**.

The word *instance* suggests that they are closely connected to the objects (which are class instances), not to the classes themselves. Let's take a closer look at them.

Here is an example:

```
class ExampleClass:
	def __init__(self, val = 1):
		self.first = val

	def setSecond(self, val):
		self.second = val

exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)

exampleObject2.setSecond(3)

exampleObject3 = ExampleClass(4)
exampleObject3.third = 5

print(exampleObject1.__dict__)
print(exampleObject2.__dict__)
print(exampleObject3.__dict__)
```

It needs one additional explanation before we go into any more detail. Take a look at the last three lines of the code.

Python objects, when created, are gifted with a **small set of predefined properties and methods**. Each object has got them, whether you want them or not. One of them is a variable named `__dict__` (it's a dictionary).

The variable contains the names and values of all the properties (variables) the object is currently carrying. Let's make use of it to safely present an object's contents.

Let's dive into the code now:

- the class named `ExampleClass` has a constructor, which **unconditionally creates an instance variable** named `first`, and sets it with the value passed through the first argument (from the class user's perspective) or the second argument (from the constructor's perspective); note the default value of the parameter - any trick you can do with a regular function parameter can be applied to methods, too;
- the class also has a **method which creates another instance variable**, named `second`;

- we've created three objects of the class `ExampleClass`, but all these instances differ:
  - `exampleObject1` only has the property named `first`;
  - `exampleObject2` has two properties: `first` and `second`;
  - `exampleObject3` has been enriched with a property named `third` just on the fly, outside the class's code - this is possible and fully permissible.

The program's output clearly shows that our assumptions are correct - here it is:

```
{'first': 1} {'second': 3, 'first': 2} {'third': 5, 'first': 4}
```

There is one additional conclusion that should be stated here: **modifying an instance variable of any object has no impact on all the remaining objects**. Instance variables are perfectly isolated from each other.

Take a look at the modified example below.

```
class ExampleClass:
    def __init__(self, val = 1):
        self.__first = val

    def setSecond(self, val = 2):
        self.__second = val


exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)

exampleObject2.setSecond(3)

exampleObject3 = ExampleClass(4)
exampleObject3.__third = 5


print(exampleObject1.__dict__)
print(exampleObject2.__dict__)
print(exampleObject3.__dict__)
```

It's nearly the same as the previous one. The only difference is in the property names.
We've **added two underscores (`__`)** in front of them.

As you know, such an addition makes the instance variable **private** - it becomes inaccessible from the outer world.

The actual behavior of these names is a bit more complicated, so let's run the program. This is the output:

```
{'_ExampleClass__first': 1}
```

```
{'_ExampleClass__first': 2, '_ExampleClass__second': 3}
{'_ExampleClass__first': 4, '__third': 5}
```

Can you see these strange names full of underscores? Where did they come from?

When Python sees that you want to add an instance variable to an object and you're going to do it inside any of the object's methods, it **mangles the operation** in the following way:

- it puts a class name before your name;
- it puts an additional underscore at the beginning.

This is why the `__first` becomes `_ExampleClass__first`.

The name is now fully accessible from outside the class. You can run a code like this:

```
print(exampleObject1._ExampleClass__first)
```

and you'll get a valid result with no errors or exceptions.

As you can see, making a property private is limited.

The mangling won't work if you add an instance variable outside the class code. In this case, it'll behave like any other ordinary property.

Class variables

A class variable is **a property which exists in just one copy and is stored outside any object**.

Note: no instance variable exists if there is no object in the class; a class variable exists in one copy even if there are no objects in the class.

Class variables are created differently to their instance siblings. The example will tell you more:

```
class ExampleClass:
    counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1

exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)
exampleObject3 = ExampleClass(4)

print(exampleObject1.__dict__, exampleObject1.counter)
print(exampleObject2.__dict__, exampleObject2.counter)
print(exampleObject3.__dict__, exampleObject3.counter)
```

Look:

- there is an assignment in the first list of the class definition - it sets the variable named `counter` to `0`; initializing the variable inside the class but outside any of its methods makes the variable a class variable;

- accessing such a variable looks the same as accessing any instance attribute - you can see it in the constructor body; as you can see, the constructor increments the variable by one; in effect, the variable counts all the created objects.

Running the code will cause the following output:

```
{'_ExampleClass__first': 1} 3
{'_ExampleClass__first': 2} 3
{'_ExampleClass__first': 4} 3
```

Two important conclusions come from the example:

- class variables **aren't shown in an object's** `__dict__` (this is natural as class variables aren't parts of an object) but you can always try to look into the variable of the same name, but at the class level - we'll show you this very soon;
- a class variable **always presents the same value** in all class instances (objects).

Mangling a class variable's name has the same effects as those you're already familiar with.

Look at the example below.

```
class ExampleClass:
    __counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.__counter += 1

exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)
exampleObject3 = ExampleClass(4)

print(exampleObject1.__dict__, exampleObject1._ExampleClass__counter)
print(exampleObject2.__dict__, exampleObject2._ExampleClass__counter)
print(exampleObject3.__dict__, exampleObject3._ExampleClass__counter)
```

Can you guess its output?

Run the program and check if your predictions were correct. Everything works as expected, doesn't it?

We told you before that class variables exist even when no class instance (object) had been created.

Now we're going to take the opportunity to show you **the difference between these two** `__dict__` **variables**, the one from the class and the one from the object.

Look at the code.

```
class ExampleClass:
    varia = 1
    def __init__(self, val):
        ExampleClass.varia = val

print(ExampleClass.__dict__)
exampleObject = ExampleClass(2)

print(ExampleClass.__dict__)
print(exampleObject.__dict__)
```

The proof is there.

Let's take a closer look at it:

- we define one class named `ExampleClass`;
- the class defines one class variable named `varia`;
- the class constructor sets the variable with the parameter's value;
- naming the variable is the most important aspect of the example because:
  - changing the assignment to `self.varia = val` would create an instance variable of the same name as the class's one;
  - changing the assignment to `varia = val` would operate on a method's local variable; (we strongly encourage you to test both of the above cases - this will make it easier for you to remember the difference)
- the first line of the off-class code prints the value of the `ExampleClass.varia` attribute; note - we use the value before the very first object of the class is instantiated.

Run the code in the editor and check its output.

As you can see, the class' `__dict__` contains much more data than its object's counterpart. Most of them are useless now - the one we want you to check carefully shows the current `varia` value.

Note that the object's `__dict__` is empty - the object has no instance variables.

Checking an attribute's existence

Python's attitude to object instantiation raises one important issue - in contrast to other programming languages, **you may not expect that all objects of the same class have the same sets of properties**.

Just like in this example. Look at it carefully.

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1

exampleObject = ExampleClass(1)

print(exampleObject.a)
print(exampleObject.b)
```

The object created by the constructor can have only one of two possible attributes: `a` or `b`.

Executing the code will produce the following output:

```
1
Traceback (most recent call last):
    File ".main.py", line 11, in
        print(exampleObject.b)
AttributeError: 'ExampleClass' object has no attribute 'b'
```

As you can see, accessing a non-existing object (class) attribute causes an `AttributeError` exception.

The `try-except` instruction gives you the chance to avoid issues with non-existent properties.

It's easy - look at the code.

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1

exampleObject = ExampleClass(1)
print(exampleObject.a)

try:
    print(exampleObject.b)
except AttributeError:
    pass
```

As you can see, this action isn't very sophisticated. Essentially, we've just swept the issue under the carpet.

Fortunately, there is one more way to cope with the issue.

Python provides a **function which is able to safely check if any object/class contains a specified property**. The function is named `hasattr`, and expects two arguments to be passed to it:

- the class or the object being checked;

- the name of the property whose existence has to be reported (note: it has to be a string containing the attribute name, not the name alone)

The function returns `True` or `False`.

This is how you can utilize it:

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1

exampleObject = ExampleClass(1)
print(exampleObject.a)

if hasattr(exampleObject, 'b'):
    print(exampleObject.b)
```

Don't forget that the `hasattr()` function can operate on classes, too. You can use it **to find out if a class variable is available**, just like here in this example.

```
class ExampleClass:
    attr = 1

print(hasattr(ExampleClass, 'attr'))
print(hasattr(ExampleClass, 'prop'))
```

The function returns `True` if the specified class contains a given attribute, and `False` otherwise.

Can you guess the code's output? Run it to check your guesses.

And one more example - look at the code below and try to predict its output:

```
class ExampleClass:
    a = 1
    def __init__(self):
        self.b = 2

exampleObject = ExampleClass()

print(hasattr(exampleObject, 'b'))
print(hasattr(exampleObject, 'a'))
print(hasattr(ExampleClass, 'b'))
print(hasattr(ExampleClass, 'a'))
```

Were you successful? Run the code to check your predictions.

Okay, we've made it to the end of this section. In the next section we're going to talk about methods, as methods drive the objects and make them active.