# Python Essentials 2: Module 6

## The Object-Oriented Approach: classes, methods, objects and the standard objective features; exception handling, and working with files

In this module, you will learn about:

the object-oriented approach - foundations;
classes, methods, objects, and the standard objective features;
exception handling;
working with files.

## (part 4)

## Methods in detail

Let's summarize all the facts regarding the use of methods in Python classes.

As you already know, a method is a function embedded inside a class.

There is one fundamental requirement - a method is obliged to have at least one parameter (there are no such thing as parameterless methods - a method may be invoked without an argument, but not declared without parameters).

The first (or only) parameter is usually named self. We suggest that you follow the convention - it's commonly used, and you'll cause a few surprises by using other names for it.

The name self suggests the parameter's purpose - it identifies the object for which the method is invoked.

If you're going to invoke a method, you mustn't pass the argument for the self parameter - Python will set it for you.

The example in the editor shows the difference.

```
class Classy:
    def method(self):
        print("method")


obj = Classy()
obj.method()
```

The code outputs:

```
method
```
output

Note the way we've created the object - we've treated the class name like a function, returning a newly instantiated object of the class.
If you want the method to accept parameters other than self, you should:

place them after self in the method's definition;
deliver them during invocation without specifying self (as previously)

Just like here:

```
class Classy:
    def method(self, par):
        print("method:", par)


obj = Classy()
obj.method(1)
obj.method(2)
obj.method(3)
```

The code outputs:
```
method: 1
method: 2
method: 3
```
output

## Methods in detail: continued

The self parameter is used to obtain access to the object's instance and class variables.

The example shows both ways of utilizing self:

```
class Classy:
    varia = 2
    def method(self):
        print(self.varia, self.var)


obj = Classy()
obj.var = 3
obj.method()
```

The code outputs:

```
2 3
```
output

The self parameter is also used to invoke other object/class methods from inside the class.

Just like here:

```
class Classy:
    def other(self):
        print("other")

    def method(self):
        print("method")
        self.other()


obj = Classy()
obj.method()
```

The code outputs:

```
method
other
```
output

## Methods in detail: continued

If you name a method like this: __init__, it won't be a regular method - it will be a constructor.

If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated.

The constructor:

is obliged to have the self parameter (it's set automatically, as usual);

may (but doesn't need to) have more parameters than just self; if this happens, the way in which the class name is used to create the object must reflect the __init__ definition;

can be used to set up the object, i.e., properly initialize its internal state, create instance variables, instantiate any other objects if their existence is needed, etc.

Look at the code in the editor. The example shows a very simple constructor at work.

```
class Classy:
    def __init__(self, value):
        self.var = value


obj_1 = Classy("object")

print(obj_1.var)
```

Run it. The code outputs:

```
object
```
output

Note that the constructor:

cannot return a value, as it is designed to return a newly created object and nothing else;

cannot be invoked directly either from the object or from inside the class (you can invoke a constructor from any of the object's subclasses, but we'll discuss this issue later.)

## Methods in detail: continued

As __init__ is a method, and a method is a function, you can do the same tricks with constructors/methods as you do with ordinary functions.

The example in the editor shows how to define a constructor with a default argument value. Test it.

```
class Classy:
    def __init__(self, value = None):
        self.var = value


obj_1 = Classy("object")
obj_2 = Classy()

print(obj_1.var)
print(obj_2.var)
```

The code outputs:

```
object
None
```
output

Everything we've said about property name mangling applies to method names, too - a method whose name starts with __ is (partially) hidden.

The example shows this effect:

```
class Classy:
    def visible(self):
        print("visible")

    def __hidden(self):
        print("hidden")


obj = Classy()
obj.visible()

try:
    obj.__hidden()
except:
    print("failed")

obj._Classy__hidden()
```

The code outputs:

```
visible
failed
hidden
```
output
Run the program, and test it.

## The inner life of classes and objects

Each Python class and each Python object is pre-equipped with a set of useful attributes which can be used to examine its capabilities.

You already know one of these - it's the __dict__ property.

Let's observe how it deals with methods - look at the code in the editor.

```
class Classy:
    varia = 1
    def __init__(self):
        self.var = 2

    def method(self):
        pass

    def __hidden(self):
        pass


obj = Classy()

print(obj.__dict__)
print(Classy.__dict__)
```

Run it to see what it outputs. Check the output carefully.

```
{'var': 2}
{'__module__': '__main__', 'varia': 1, '__init__': <function Classy.__init__ at
0x7f44070b6320>, 'method': <function Classy.method at 0x7f44070b63b0>,
'_Classy__hidden': <function Classy.__hidden at 0x7f44070b6440>, '__dict__':
<attribute '__dict__' of 'Classy' objects>, '__weakref__': <attribute '__weakref__' of
'Classy' objects>, '__doc__': None}
```

Find all the defined methods and attributes. Locate the context in which they exist: inside the object or inside the class.

## The inner life of classes and objects: continued

__dict__ is a dictionary. Another built-in property worth mentioning is __name__, which is a string.

The property contains the name of the class. It's nothing exciting, just a string.

Note: the __name__ attribute is absent from the object - it exists only inside classes.

If you want to find the class of a particular object, you can use a function named type(), which is able (among other things) to find a class which has been used to instantiate any object.

Look at the code in the editor, run it, and see for yourself.

```
class Classy:
    pass


print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)
```

The code outputs:

```
Classy
Classy
```
output

Note that a statement like this one:

```
print(obj.__name__)
```

will cause an error.

## The inner life of classes and objects: continued

__module__ is a string, too - it stores the name of the module which contains the definition of the class.

Let's check it - run the code in the editor.

```
class Classy:
    pass


print(Classy.__module__)
obj = Classy()
print(obj.__module__)
```

The code outputs:

```
__main__
__main__
```

output

As you know, any module named __main__ is actually not a module, but the file currently being run.

## The inner life of classes and objects: continued

__bases__ is a tuple. The tuple contains classes (not class names) which are direct superclasses for the class.

The order is the same as that used inside the class definition.

We'll show you only a very basic example, as we want to highlight how inheritance works.

Moreover, we're going to show you how to use this attribute when we discuss the objective aspects of exceptions.

Note: only classes have this attribute - objects don't.

We've defined a function named printbases(), designed to present the tuple's contents clearly.

Look at the code in the editor.

```
class SuperOne:
    pass


class SuperTwo:
    pass


class Sub(SuperOne, SuperTwo):
    pass


def printBases(cls):
    print('( ', end='')

    for x in cls.__bases__:
        print(x.__name__, end=' ')
    print(')')


printBases(SuperOne)
printBases(SuperTwo)
printBases(Sub)
```

Analyze it and run it. It will output:

```
( object )
( object )
( SuperOne SuperTwo )
```
output

Note: a class without explicit superclasses points to object (a predefined Python class) as its direct ancestor.

## Reflection and introspection

All these means allow the Python programmer to perform two important activities specific to many objective languages. They are:

introspection, which is the ability of a program to examine the type or properties of an object at runtime;

reflection, which goes a step further, and is the ability of a program to manipulate the values, properties and/or functions of an object at runtime.

In other words, you don't have to know a complete class/object definition to manipulate the object, as the object and/or its class contain the metadata allowing you to recognize its features during program execution.

# Investigating classes

What can you find out about classes in Python? The answer is simple - everything.

Both reflection and introspection enable a programmer to do anything with every object, no matter where it comes from.

Analyze the the code in the editor.

```
class MyClass:
    pass


obj = MyClass()
obj.a = 1
obj.b = 2
obj.i = 3
obj.ireal = 3.5
obj.integer = 4
obj.z = 5


def incIntsI(obj):
    for name in obj.__dict__.keys():
        if name.startswith('i'):
            val = getattr(obj, name)
            if isinstance(val, int):
                setattr(obj, name, val + 1)


print(obj.__dict__)
incIntsI(obj)
print(obj.__dict__)
```

The function named incIntsI() gets an object of any class, scans its contents in order to find all integer attributes with names starting with i, and increments them by one.

Impossible? Not at all!

This is how it works:

line 1: define a very simple class...
lines 3 through 10: ... and fill it with some attributes;
line 12: this is our function!
line 13: scan the __dict__ attribute, looking for all attribute names;
line 14: if a name starts with i...
line 15: ... use the getattr() function to get its current value; note: getattr() takes two arguments: an object, and its property name (as a string), and returns the current attribute's value;
line 16: check if the value is of type integer, and use the function isinstance() for this purpose (we'll discuss this later);
line 17: if the check goes well, increment the property's value by making use of the setattr() function; the function takes three arguments: an object, the property name (as a string), and the property's new value.

The code outputs:

```
{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}
{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}
output
```

That's all!

Key takeaways

1. A method is a function embedded inside a class. The first (or only) parameter of each method is usually named self, which is designed to identify the object for which the method is invoked in order to access the object's properties or invoke its methods.

2. If a class contains a constructor (a method named __init__) it cannot return any value and cannot be invoked directly.

3. All classes (but not objects) contain a property named __name__, which stores the name of the class. Additionally, a property named __module__ stores the name of the module in which the class has been declared, while the property named __bases__ is a tuple containing a class's superclasses.

For example:

```
class Sample:
    def __init__(self):
        self.name = Sample.__name__
    def myself(self):
        print("My name is " + self.name + " living in a " + Sample.__module__)

obj = Sample()
obj.myself()
```

The code outputs:

```
My name is Sample living in a __main__
```
output

Exercise 1

The declaration of the Snake class is given below. Enrich the class with a method named increment(), adding 1 to the __victims property.

```
class Snake:
    def __init__(self):
        self.victims = 0
```

Exercise 2

Redefine the Snake class constructor so that is has a parameter to initialize the victims field with a value passed to the object during construction.

Exercise 3

Can you predict the output of the following code?

```
class Snake:
    pass

class Python(Snake):
    pass

print(Python.__name__, 'is a', Snake.__name__)
print(Python.__bases__[0].__name__, 'can be', Python.__name__)
```