1.1 Classes, Instances, Attributes, Methods — introduction

Introduction to Object-Oriented Programming

This module addresses the advanced Object Oriented Programming (OOP) issues that are at the heart of Python programming.

The object-oriented approach is an evolution of good design practices that go back to the very beginning of computer programming.

This very important approach is present in most computer applications because it allows programmers to model entities representing real-life objects. Moreover, OOP allows programmers to model interactions between objects in order to solve real-life problems in an efficient, comfortable, extendable, and well-structured manner.

Imagine that the text you are now reading was rendered in a web browser created with OOP, and even the mouse cursor image is displayed using an application created with OOP. The same applies to spreadsheet handling (treat a spreadsheet as a collection of objects), and many, many other applications.

This chapter assumes that you are familiar with the basics of OOP, so to establish an understanding of common terms, we should agree on the following definitions:

- class — an idea, blueprint, or recipe for an instance;
- instance — an instantiation of the class; very often used interchangeably with the term 'object';
- object — Python's representation of data and methods; objects could be aggregates of instances;
- attribute — any object or class trait; could be a variable or method;
- method — a function built into a class that is executed on behalf of the class or object; some say that it’s a 'callable attribute';
- type — refers to the class that was used to instantiate the object.

Now that we’re starting to discuss more advanced OOP topics, it’s important to remember that in Python everything is an object (functions, modules, lists, etc.). In the very last section of this module, you'll see that even classes are instances.

Why is everything in Python organized as objects?

Because an object is a very useful culmination of all the terms described above:

- it is an independent instance of class, and it contains and aggregates some specific and valuable data in attributes relevant to individual objects;
- it owns and shares methods used to perform actions.

The following issues will be addressed during this and the next module:

- creation and use of decorators;
- implementation of core syntax;
- class and static methods;
- abstract methods;
- comparison of inheritance and composition;
- attribute encapsulation;
- exception chaining;
- object persistence;
- metaprogramming.

class

A class expresses an idea; it’s a blueprint or recipe for an instance. The class is something virtual, it can contain lots of different details, and there is always one class of any given type. Think of a class as a building blueprint that represents the architect’s ideas, and class instances as the actual buildings.

Classes describe attributes and functionalities together to represent an idea as accurately as possible.

You can build a class from scratch or, something that is more interesting and useful, employ inheritance to get a more specialized class based on another class.

Additionally, your classes could be used as superclasses for newly derived classes (subclasses).

Python’s class mechanism adds classes with a minimum of new syntax and semantics:

```
class Duck:
    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex

    def walk(self):
        pass

    def quack(self):
        return print('Quack')
```

In the code above, we have defined a class named Duck, consisting of some functionalities and attributes.

A class is a place which binds data with the code.

If you run the code, there are no visible effects. The class has been defined, but there is no code making use of it — that’s why you see no effects.

instance

An instance is one particular physical instantiation of a class that occupies memory and has data elements. This is what 'self' refers to when we deal with class instances.

An object is everything in Python that you can operate on, like a class, instance, list, or dictionary.

The term instance is very often used interchangeably with the term object, because object refers to a particular instance of a class. It’s a bit of a simplification, because the term object is more general than instance.

The relation between instances and classes is quite simple: we have one class of a given type and an unlimited number of instances of a given class.

Each instance has its own, individual state (expressed as variables, so objects again) and shares its behavior (expressed as methods, so objects again).

To create instances, we have to instantiate the class:

```
duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
```

```
hen = Duck(height=20, weight=3.4, sex="female")
```

n the example presented above, we have created three different instances based on the **Duck** class: duckling, drake and hen. We haven't called any object attributes.

attribute

An attribute is a capacious term that can refer to two major kinds of class traits:

- variables, containing information about the class itself or a class instance; classes and class instances can own many variables;
- methods, formulated as Python functions; they represent a behavior that could be applied to the object.

Each Python object has its own individual set of attributes. We can extend that set by adding new attributes to existing objects, change (reassign) them or control access to those attributes.

It is said that methods are the 'callable attributes' of Python objects. By 'callable' we should understand anything that can be called; such objects allow you to use round parentheses () and eventually pass some parameters, just like functions.

This is a very important fact to remember: methods are called on behalf of an object and are usually executed on object data.

Class attributes are most often addressed with 'dot' notation, i.e., <class>dot<attribute>. The other way to access attributes (variables) it to use the getattr() and setattr() functions.

In our 'duckish' example, there are the following attributes:

- variables: self.height, self.weight, self.sex — containing different values for each object;
- methods: __init__, walk, quack — common to all objects so far.

Examples:

- To call a method, issue: drake.quack();
- To access an attribute, issue: print(duckling.height).

If you run the code, you'll get the following example:

```
class Duck:
    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex

    def walk(self):
        pass

    def quack(self):
        return print('Quack')

duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")

drake.quack()
print(duckling.height)
```

output

```
Quack
10
```

type

A type is one of the most fundamental and abstract terms of Python:

- it is the foremost type that any class can be inherited from;
- as a result, if you're looking for the type of class, then type is returned;
- in all other cases, it refers to the class that was used to instantiate the object; it's a general term describing the type/kind of any object;
- it's the name of a very handy Python function that returns the class information about the objects passed as arguments to that function;
- it returns a new type object when type() is called with three arguments; we'll talk about this in the 'metaclass' section.

Python comes with a number of built-in types, like numbers, strings, lists, etc., that are used to build more complex types. Creating a new class creates a new type of object, allowing new instances of that type to be made.

Information about an object's class is contained in **__class__**.

If you run the code presented in the right pane, you'll get the type details of different objects.

```
class Duck:
    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex

    def walk(self):
        pass

    def quack(self):
        return print('Quack')

duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")

print(Duck.__class__)
print(duckling.__class__)
print(duckling.sex.__class__)
print(duckling.quack.__class__)
```

output

```
<class 'type'>
<class '__main__.Duck'>
<class 'str'>
```

```
<class 'method'>
```

As we predicted:

- the **Duck** class is of the 'type' type;
- the **duckling** object is an instance type built on the basis of the **Duck** class, and residing in the **__main__** scope;
- the **duckling.sex** is an attribute of the 'str' type;
- **duckling.quack** is an attribute of the 'method' type.

LAB

Python allows for variables to be used at the instance level or the class level. Those used at the instance level are referred to as **instance variables**, whereas variables used at the class level are referred to as **class variables**.

Scenario

- create a class representing a mobile phone;
- your class should implement the following methods:
  - __init__ expects a number to be passed as an argument; this method stores the number in an instance variable self.number
  - turn_on() should return the message 'mobile phone {number} is turned on'. Curly brackets are used to mark the place to insert the object's number variable;
  - turn_off() should return the message 'mobile phone is turned off';
  - call(number) should return the message 'calling {number}'. Curly brackets are used to mark the place to insert the object's number variable;
- create two objects representing two different mobile phones; assign any random phone numbers to them;
- implement a sequence of method calls on the objects to turn them on, call any number. Print the methods' outcomes;
- turn off both mobiles.

output

```
mobile phone 01632-960004 is turned on
mobile phone 01632-960012 is turned on
calling 555-34343
mobile phone is turned off
mobile phone is turned off
```

respuesta

```
class mobil:
    def __init__(self, number):
        self.number = number

    def turn_on(self):
        print("el mòbil %s està engegat" % (self.number) )

    def turn_off(self):
        print("el mòbil %s està apagat" % (self.number) )

    def call(self,number):
        print("trucant al %s" % number )

m1 = mobil("111.111.111")
m2 = mobil("222.222.222")

m1.turn_on()
m2.turn_on()
m1.call("666.666.666")
m1.turn_off()
m2.turn_off()
```

From:
https://miguelangel.torresegea.es/wiki/ - **miguel angel torres egea**

Permanent link:
https://miguelangel.torresegea.es/wiki/info:cursos:pue:python-pcpp1:m1:1.1

Last update: **05/11/2023 12:31**