“Object Oriented Programming.”

Sun May 09 2021

Paradigms of software programming:

When designing a software system, various thought patterns can be used.

- Imperative paradigm: calling functions, using variable assignments.
- Functional paradigm: calling functions that call functions, without ever using variable assignments.
- Object oriented paradigm: defining software entities that offer services to other entities.

Python is a language that will let us use any of the 3 paradigms. But in this course, we'll focus on what is the Object Oriented paradigm and how to use it with Python.

Object Oriented Programming:

Object-oriented Programming (OOP) is a programming paradigm based on the concept of “objects”. Each object can contain data and operations: data in the form of fields or attributes or properties; and operations, in the form of procedures or methods.

Implementing a system using the object-oriented paradigm will look something like:

- Identifying objects in the physical world.
- Defining or modeling the internals of the objects, the attributes.
- Defining or modeling the behavior of the objects, the methods.
- Describing how the objects interact.

There are four principles of Object Oriented Programming. If all of these four principles are present in a programming language, the language is said to allow Object Oriented Programming. Those four principles are:

- Inheritance
- Encapsulation
- Polymorphism
- Abstraction

Object concept:

Objects sometimes correspond *more or less* to entities found in the physical world. For example, an online shopping system might have objects such as “shopping cart”, “product”, “customer”, and “invoice”. Other example, a software may have the objects such as “window”, “button”, “form”, “text field”, “menu”.

An **Object** is a software entity that has:

- a unique identifier
- a set of **attributes** containing information
  - each attribute can have a different type (integer, real, string, list, …)
  - an attribute may be public or private, i.e. accessible or not from the oustide

- a set of **operations** that manipulate the object's attributes or offer services
  - an operation may be public or private

Class concept:

Each object is said to be an instance of a particular class. A class is a template definition of the attributes and the methods a particular object should have. In other words, a class is the blue print from which an individual object is created.

A **class** is a software entity that has:

- a unique **name**
- a set of **attributes**
  - these are the attributes that will be available into each object built from this class
  - each attribute has a unique name (inside the context of the class)
  - each attribute has a type
- a set of **operations**
  - these are the operations that will be available for the objects built from this class
  - each operation has a unique **signature** (in the context of the class)
    * the signature contains the name of the operation, the type of the result it returns
    * the signature contains the various parameters given to the operation with their respective types

Now, let's explore how all those concept are implemented in Python.

Classes in Python:

In Python, everything is an object. In the following example, we can see, `a` is a variable of the integer type. This object has *at least* 2 attributes: `real` and `imag`. This object has *at least* 2 operations: `conjugate` and `as_integer_ratio`:

```
>>>
>>> a = 3
>>> type(a)
<class 'int'>
>>>
>>> a.imag
0
>>> a.real
3
>>> a.conjugate()
3
>>> a.as_integer_ratio()
(3, 1)
>>>
```

Everytime you manipulated some variables, in fact, you manipulated some objects created using some classes. You can double check by:

```
>>> help(int)

>>> help(str)

>>> help(complex)

>>> help(float)
```

Create your own types:

Time will come when you'll need to manipulate objects of the physical world inside your program. For being able to define our own classes and create our own objects, Python gives us a way.

How to create a class:

The following is how you create a very simple class, a class that does nothing:

```
>>> # definition la plus simple pouvant exister
>>> # Premiere lettre en majuscule
>>> # toujours au singulier, jamais au pluriel
>>> # definir une classe nommée Program
>>> # qui ne fait absolument rien du tout
>>>
>>> class Foobar:
...     pass
...
>>>
```

A class may contain nothing but methods. This method may take any number of parameters. But the convention tells us: `self` should always be the first parameter. Reminder: a method cannot be called outside of the class context. Example:

```
>>> # definition d'une classe nommée Barbaz
>>> # cette classe aura 2 méthodes: f et g
>>> # f est une méthode ne prenant aucun parametre
>>> # g est une méthode prenant un parametre nommé name
>>> class Barbaz:
>>>     """A simple example class"""
>>>
>>>     def f(self):
>>>         return 'hello world'
>>>
>>>     def g(self, name="qux"):
>>>         qux = name
>>>         return f'hello {qux}'
```

A class may also contains nothing but attributes. If you want attributes in your class, you'll need to use a special method name "dunder init": `__init__`. Example:

```
>>> # definition d'une classe nommée Quux
>>> # cette classe aura 3 attributs: name, age, address
>>> class Quux:
>>>     """A simple example class"""
>>>
>>>     def __init__(self, name, age, address):
>>>         self.name = name
>>>         self.age = age
>>>         self.address = address
```

Finally, a class may contains, attributes **and** methods, for example:

```
>>> class Dog:
>>>     def __init__(self, name):
>>>         # 2 attributes
>>>         self.name = name
>>>         self.tricks = []
>>>
>>>     # one method
>>>     def add_trick(self, trick):
>>>         self.tricks.append(trick)


>>> class Bag:
>>>     def __init__(self):
>>>         # one attribute
>>>         self.data = []
>>>
>>>     # the first method
>>>     def add(self, x):
>>>         self.data.append(x)
>>>
>>>     # the second method
>>>     def addtwice(self, x):
>>>         self.add(x)
>>>         self.add(x)
```

How to instanciate objects:

Creating classes is interesting. The main role of our class is to create (to instanciate) objects. To instanciate an object, you'll do as if you're calling a function. For example:

```
>>> a = Foobar()
>>> type(a)
>>> isinstance(a, Foobar)
>>> help(a)
```

Another example:

```
>>> b = Barbaz()
>>> type(b)
>>> isinstance(b, Foobar)
>>> isinstance(b, Barbaz)
>>> dir(b)
>>> help(b)
>>>
>>> c = Quux("patrick", 42, "Dakar")
>>> dir(c)
```

In the following example, we see that attributes can be assigned **dynamically**, not just during initialization:

```
>>> f = Foobar()
>>> f.age = 24
>>> f.name = "Fatimata"
>>> f.nick = "Fatim"
>>> type(f)
>>> isinstance(f, Foobar)
```

```
>>>
>>> print("{0} is {1} years old, and her nickname is {2}".format(f.name, f.age, f.nick))
```

What is inheritance:

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that child classes are derived from are called parent classes. Compare the following code:

```
>>> class Lemur:
>>>     def __init__(head, arms, fingers, locomotion, vision):
>>>         self.head = head
>>>         self.arms = arms
>>>         self.fingers = fingers
>>>         self.locomotion = locomotion
>>>         self.vision = vision
>>>
>>>     def move(self):
>>>         print("I'm moving myself")
>>>
>>>     def eat(self):
>>>         print("I'm eating fruits")
>>>
>>> class Monkey:
>>>     def __init__(head, arms, fingers, locomotion, vision):
>>>         self.head = head
>>>         self.arms = arms
>>>         self.fingers = fingers
>>>         self.locomotion = locomotion
>>>         self.vision = vision
>>>
>>>     def move(self):
>>>         print("I'm moving myself")
>>>
>>>     def eat(self):
>>>         print("I'm eating fruits")
>>>
>>> class Human:
>>>     def __init__(head, arms, fingers, locomotion, vision):
>>>         self.head = head
>>>         self.arms = arms
>>>         self.fingers = fingers
>>>         self.locomotion = locomotion
>>>         self.vision = vision
>>>
>>>     def move(self):
>>>         print("I'm moving myself")
>>>
>>>     def eat(self):
>>>         print("I'm eating fruits")
>>>
```

To this:

```
>>> class Primate:
>>>     def __init__(head, arms, fingers, locomotion, vision):
>>>         self.head = head
>>>         self.arms = arms
>>>         self.fingers = fingers
>>>         self.locomotion = locomotion
>>>         self.vision = vision
>>>
>>>     def move(self):
>>>         print("I'm moving myself")
>>>
>>>     def eat(self):
>>>         print("I'm eating fruits")

>>> class Lemur(Primate):
>>>     pass

>>> class Monkey(Primate):
>>>     pass

>>> class Human(Primate):
>>>     pass
```

Thanks to inheritance, we managed to keep the features in one class, reuse all those features in the other classes and write less code.

It's perfectly possible for a child class, to implement its own methods:

```
>>>
>>> class Fish:
>>>     def __init__(self, name, age, skeleton, eyelids):
>>>         self.name = name
>>>         self.age = age
>>>         self.skeleton = skeleton
>>>         self.eyelids = eyelids
>>>
>>>     def swim(self):
>>>         print("I'm swimming.")
>>>
>>>
>>> class Trout(Fish):
>>>
>>>      # cette méthode n'existe pas dans la classe parente
>>>      def swim_backwards(self):
>>>          print("I can swim backwards.")
```

It's perfectly possible for a class, to implement its own attributes, thanks to the **super** function:

```
In [8]: class Person:
   ...:     def __init__(self, name, phone):
   ...:         self.name = name
   ...:         self.phone = phone
   ...:

In [9]: class Teenager(Person):
   ...:     def __init__(self, name, phone, website):
   ...:         # thanks to super, we can call the parent class constructor
   ...:         # and initiate the name & phone attributes
   ...:         super().__init__(name, phone)
   ...:         self.website=website
   ...:
   ...:
```

Extending Methods:

This is the way to say "do everything the parent methods does, **plus** this other stuff". Here again, we'll use the **super** function:

```
>>>
>>> class Fish:
>>>     def __init__(self, name, age, skeleton, eyelids):
>>>         self.name = name
>>>         self.age = age
>>>         self.skeleton = skeleton
>>>         self.eyelids = eyelids
>>>
>>>     # methode swim définie dans la classe parente
>>>     def swim(self):
>>>         print("I'm swimming.")
>>>
>>>
>>> class Trout(Fish):
>>>
>>>     def swim_backwards(self):
>>>         print("I can swim backwards.")
>>>
>>>     def swim_like_a_fish_and_more(self):
>>>         print("*¨°□@o.")
>>>         super().swim() # super nous permet d'appeler swim de la classe parente
>>>         print(".o0@□°¨*")
>>>
```

What is encapsulation:

Encapsulation is the mechanism of restricting access to public methods or public attributes. By default, our attribtues and our methods are public, i.e. you can access them directly without any restrictions: `obj.attribute` or `obj.method()`. If we want our attributes or our methods to be private, we just have to name them using 2 leading underscores. Example:

```
>>> class A:
>>>     def __init__(self):
>>>         self.public_var = "abc"
>>>         self.__private_var = 123
>>>
>>>     def printVar(self):
>>>         # from inside the class we still can access the attribute, even if private
>>>         print(self.__private_var)
>>>
```

One use of encapsulation is to hide the implementation details of some service. For example, in the following class, we want to draw the face of Felix. To do saw, the service is offered via a public method: `draw_felix`. Our engineers decided to implement this service using two steps: draw the eyes, then draw the smile. Do the final users need to have access to those 2 methods? Of course not. The final user just need an access to the service: draw_felix:

```
In [30]: class Cat:
    ...:     def __draw_eyes(self):
    ...:         eyes = """
    ...: .:.
    ...:              .::::.
    ...: ..         ..::::::''::
    ...: ::::..  .::''''':::    ''.
    ...: ':::::::'         '.  ..  '.
    ...:  ::::::'            : '::   :
    ...:   :::::     .        : ':'   :
    ...:   :::::    :::       :.     .'.
    ...:  .::::::    ':'     .' '.:::: :
    ...:  ::::::::.         .    ::::: :
    ...: :::::    '':.... ''      '''' : """
    ...:         print(eyes, end='')
    ...:
    ...:     def __draw_smile(self):
    ...:         smile = """
    ...: '::::  .:'               ...'' :
    ...:  ..::.   '.........:::::'    :
    ...:   '':::.   '::'':'':::'    .'
    ...:         '..  ''.....'  ..'
    ...:            ''........''
    ...: """
    ...:         print(smile)
    ...:
    ...:     def draw_felix(self):
    ...:         self.__draw_eyes()
    ...:         self.__draw_smile()
    ...:
    ...:
    ...:
```

What is polymorphism:

Polymorphism is the process of using an operator or function in different ways for different data input. In the following example, notice how the len function is able to work despite the fact that arguments passed are of different types:

```
In [38]: a = "Programiz"

In [39]: b = ["Python", "Java", "C"]

In [40]: c = {"Name": "John", "Address": "Nepal"}

In [41]: my_list = [a, b, c]

In [42]: for x in my_list:
    ...:     print(len(x))
    ...:
9
3
2

In [43]:
```

Python allows different classes to have methods with the same name. Example:

```
In [43]: class Cat:
    ...:     def __init__(self, name):
    ...:         self.name = name
    ...:
    ...:     def make_sound(self):
    ...:         print("Meow")
    ...:
    ...: class Dog:
    ...:     def __init__(self, name):
    ...:         self.name = name
    ...:
    ...:     def make_sound(self):
    ...:         print("Bark")
    ...:
    ...:

In [44]: dog = Dog("Bobby")

In [45]: cat = Cat("Kitty")

In [46]: for animal in [dog, cat]:
    ...:     animal.make_sound()
    ...:
Bark
Meow

In [47]:
```

You may have already guessed it. Polymorphism can be achieved through inheritance. Example:

```
   ...: class Shape:
    ...:     def __init__(self, name):
    ...:         self.name = name
    ...:
    ...:     def area(self): # there is no implementation in the parent class
    ...:         pass
    ...:
    ...:     def __str__(self): # this is another magic method
    ...:         return self.name
    ...:

In [48]:

In [48]: class Square(Shape):
    ...:     def __init__(self, length):
    ...:         super().__init__("Square")
    ...:         self.length = length
    ...:
    ...:     # we override a method from the parent class
    ...:     def area(self):
    ...:         return self.length**2
    ...:

In [49]: class Circle(Shape):
    ...:     def __init__(self, radius):
    ...:         super().__init__("Circle")
    ...:         self.radius = radius
    ...:
    ...:     # another overriding of the area method
    ...:     def area(self):
    ...:         return pi*self.radius**2
    ...:

In [50]: s = Square(4)

In [51]: c = Circle(6)

In [53]: for x in [c, s]:
    ...:     print(x.area())
    ...:
    ...:
113.09733552923255
16

In [54]:
```

What is abstraction:

Abstraction is the way to express the intent of the class rather than the actual implementation.

Let's suppose we writing a program which models a house. The **house** will have several different **rooms** (our objects). We'll define a **BedRoom** class, a **Kitchen** class, a **BathRoom** class, a **LivingRoom**, a **DiningRoom**, etc…

All this rooms will **share several properties** (number of doors, number of windows, square feet, …). So we can use **inheritance**, create a **Room** class, and make LivingRoom, DiningRoom, Bedroom, Kitchen, BathRroom classes inherit from Room. Now there is a **question: is it possible for a Room object to exists on its own?** The answer is "no".

It makes sense to create **an abstract class** called Room, which will contain the properties all rooms share, and then have the *usefull* classes, inherit from the abstract class Room.

The concept of a room is **abstract** and only exists in our head, because any room that actually exists isn't just a room; it's a bedroom or a living room or a kitchen, etc…

To recap:

- An abstract class cannot be instantiated.
- An abstract class contains one or more abstract methods (with no detailed implementation).
- A derived subclass must implement the abstract methods defined in the abstract class
- A object from the derived class cannot be instantiated unless all of its abstract methods are overridden.

```
In [75]: from abc import ABC, abstractmethod
    ...:
    ...: class Room(ABC):
    ...:
    ...:     def __init__(self, door_nbr, window_nbr):
    ...:         self.door_nbr = door_nbr
    ...:         self.window_nbr = window_nbr
    ...:
    ...:     @abstractmethod
    ...:     def area(self):
    ...:         pass
    ...:

In [76]: class BedRoom(Room):
    ...:     pass
    ...:
    ...:
    ...:

In [77]: bedroom1 = BedRoom(2, 2)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-77-458ab6c446b5> in <module>
----> 1 bedroom1 = BedRoom(2, 2)

TypeError: Can't instantiate abstract class BedRoom with abstract methods area

In [78]:
```

```
In [81]: class BedRoom(Room):
    ...:     def area(self):
    ...:         return "The bedroom area is 25 square feets"
    ...:

In [82]:
```

```
In [82]: bedroom1 = BedRoom(2, 2)

In [83]: bedroom1.area()
Out[83]: 'The bedroom area is 25 square feets'

In [84]: bedroom1.door_nbr
Out[84]: 2

In [85]: bedroom1.window_nbr
Out[85]: 2

In [86]:
```