Introduction to Programming W14

Introduction to Object-Oriented Programming in Python I.

2022/07/08

Object-Oriented Programming (OOP) is a programming paradigm supported by Python and many other languages as well. In OOP, programs are structured in a way that *attributes* and *behaviors* are bundled into *objects*. For example, an object `Cat` has attributes like name, size, fur pattern, color, etc., and behavior like meowing, purring, eating, etc. Another way to look at it is that OOP is a method for modeling things and the relation between things (e.g., teachers and students). In OOP, such entities are represented as software objects that can perform some predetermined functions. Practically speaking, OOP focuses on creating reusable code.

1 Classes, Instances, Objects

You can think of a **class** as a blueprint for an object: e.g. it includes the attributes, and identifies the behaviors the class can perform. The `Cat` class defines that name, color, etc. are needed for defining a cat, but it does not contain a specific name or color of a specific cat. An **instance** is an object that is an instantiation of a class: it is built from a class using actual data. For example, an instance of the `Cat` class is a cat with the name Marie who is orange colored. Another way to look at the class-instance relationship is thinking about filling in a survey/questionnaire: multiple people can fill the same survey form (class) with their own unique answers, creating multiple filled out forms (instances).

Practically speaking, classes are used to create user-defined data structures. Class definitions start with the `class` keyword followed by the name of the class and a colon. The class's body is indented below the class definition. Class names in Python are written in CapitalizedWords by convention.

```
class Cat:
    pass
```

Here, the body of the `Cat` class only has a single `pass` inside, which is just a placeholder indicating places where code will go eventually (you may use the `pass` keyword as a placeholder in functions as well). The attributes that all `Cat` objects *must have* are defined in the `.__init__()` method (note the double underscores). This method initialized the instances of the class, by setting the initial state of the object: it assigns data (values) of the attributes. The method `__init__` is defined just as a normal function, but the first argument it takes must be a variable called `self`. At the creation of a new instance, it is passed to `self` so that new attributes can be defined. `__init__` is indented inside the class, indicating it belongs to it.

```
class Cat:
    def __init__(self, name, color):
        self.name = name
        self.color = color
```

Here, `self.name = name` creates an attribute called `name`, and assigns the value of the `name` argument to it. The same way, `self.color = color` creates an attribute called `color`, and assigns the value of the `color` argument to it. Initialized class attributed created with `.__init__()` are **instance attributes**. All `Cat` objects have names and colors, but `name` and `color` attribute values are up to the specific instance. If an attribute is defined outside of the `__init__` method, are class instances will have the same value for that attribute. These are called **class attributes**.

```
class Cat:
    animal_type = "carnivore" #class attribute

    def __init__(self, name, color):
        self.name = name
        self.color = color
```

The above class has a class attribute called `animal_type`. When an instance is created, the class attributes are automatically assigned to their initial values (here, `carnivore`). So if you want to define an attribute that is the same for every instance object, you must use class attributes. If an attribute varies case-by-case, you need to use instance attributes.

2 Object instantiation

Creating an object from a class is called **instantiation**. To instantiate an object of the `Cat` class, one needs to provide values for the instance attributes (otherwise there will be an error). Instantiation happens outside of the class definition. Try the following code:

```
class Cat:
    animal_type = "carnivore" #class attribute

    def __init__(self, name, color, age):
        self.name = name
        self.color = color
        self.age = age

marie = Cat("Marie", "orange", 9) #object instantiation
tutu = Cat("Tutu", "brown", 6) #object instantiation

print("Name of the marie object:", marie.name) #accessing instance attributes
print("Color and age of the tutu object:", tutu.color, tutu.age) #accessing instance attributes
print("Animal type of the marie object:", marie.animal_type) #accessing a class attribute

marie.age = 8 #changing the age attribute of the object marie
print("Age of the marie object (changed):", marie.age)
```

Here, two new `Cat` instances were created. At object instantiation, a new instance is created and passed to the first argument of `__init__` (which is always `self`), so only the other arguments need to be provided. Using a dot notation, all attributes can be accessed, and can be changed if needed (both class and instance attributes). This means that custom objects in Python are mutable by default.

3 Instance methods

Functions defined inside a class are called **instance methods**. You can think of them in a way that these methods specify the object's behavior. These methods can be called only from an instance of the class. Same with the `.__init__()` method, their first parameter is `self`.

```
class Cat:
    animal_type = "carnivore" #class attribute

    def __init__(self, name, color, age):
        self.name = name
        self.color = color
        self.age = age

    def describe(self):
        return "{} is {} years old".format(self.name, self.age)

    def say(self, sound):
        return "{} says {}".format(self.name, sound)

marie = Cat("Marie", "black", 3)
print(marie.describe()) #calling the describe() instance method, no arguments needed
print(marie.say("meow")) #calling the say() instance method with sound="meow"
```

4 Practice

Create a class called `Car` with two class attributes: `classification= "Vehicle"` and `application= "Transportation"`. Instance attributes should be `name`, `fuel_source` and `speed`. Create an instance method called `accelerate` that takes an integer argument `acc` (besides `self`) and returns the speed of the car defined in the instance attribute `speed` + `acc`. Create new instances of the class (e.g., `bmw = Car("BMW", "gasoline", 60)`), and make sure the `.accelerate()` method works (e.g., `print(bmw.accelerate(20))` should print `80` if the object was instantiated with speed=60).

Next class we will continue learning about OOP in Python, and your final assignment will require you to use OOP, so make sure you practice and understand today's material before the next class.