Classes and Objects

Section 22.9

Completed
< back to module

Skip straight to editor/terminal

There is a style of programming called *object-oriented* that uses a construct called a *class* to create a model of a real-world thing, and then uses that as a kind of template to create *objects* from. When we create a class, we define the general way an object is constructed and how it behaves.

For example, if we think of a real world object like a car, there are some things all cars have in common. They all have 4 wheels, doors, an engine, and a steering wheel. There are also some ways in which all cars behave similarly: they can all move forwards and backwards, they can turn left and right, and they have horns that make a noise in an emergency. If we were programming a game that needed to have lots of different cars in it, an efficient way for us to do this would be to create a *class* to model all the necessary attributes of any given car, and any actions all cars should be able to take.

In this section we aren't going to talk too much about the style of object-oriented programming specifically, but for now let's make sure we know how to create a class, and how to create objects from them.

Creating an Agent class

Let's create a very simple class that models our agents. In previous sections and modules we've learned a lot about our team of agents, so let's use some of that information to create our first class.

```
class Agent():
   name = ''
   hot_drink = ''

   def speak(self, speech):
      print(self.name + ' says: "' + speech + '"')

   def drink(self):
      print(self.name + ' drinks a cup of ' + self.hot_drink + '.')
```

Let's go through our class line by line.

We first start by declaring our class and giving it a name: `class Agent():`. Everything that belongs to our class will be indented.

Next, we create 2 variables which our class will need to work properly, but we don't assign them any values: right now, both `name` and `hot_drink` are both set to be empty strings.

After that we create 2 functions, `speak()` and `drink()`. In a class, functions are called *methods*. The only difference between a function and a method is that a method is part of a class, which means we need to call it in a slightly different way, which we'll see in the next example.

As it stands now, our `Agent()` class captures a model of a person who:

- Has a name
- Has a hot drink preference
- Can speak
- Can drink their hot beverage

Now that we've created our class, how do we use it?

```
class Agent():
   name = ''
   hot_drink = ''

   def speak(self, speech):
      print(self.name + ' says: "' + speech + '"')

   def drink(self):
      print(self.name + ' drinks a cup of ' + self.hot_drink + '.')

agent_q = Agent()
agent_q.name = 'Agent Q'
agent_q.speak("Hi, I'm Agent Q!")
```

After we've defined our class, once we want to start making use of it, the first thing we do is *instantiate* our class with `agent_q = Agent()`. Here, we create an *object* using our class as a model, and call that object `agent_q`. This object is a little self-contained "thing" that has all the attributes and methods associated to it that we defined in our class. As programmers, we would refer to this *object* as an *instance of a class*.

In the next line, we take our new object `agent_q` and set a value for the variable `name` that we left as an empty string when we created the class: `agent_q.name = 'Agent Q'`. Notice our notation here: we use *dot notation* to set the value of a class variable. Variables that are accessed on objects like this are called *attributes*.

Finally, we use one of the methods we created in our class – again, we use dot notation to access it because we are interacting with our object: `agent_q.speak("Hi, I'm Agent Q!")`.

Here is the output that's generated when we execute this code:

```
Agent Q says: "Hi, I'm Agent Q!"
```

Understanding self

Notice how we refer to the attributes we created in our class? When we want to use the `name` attribute within one of our methods, we use `self.name`. Why do we do this? The `self` argument in the class refers to itself – the object the class creates. By calling `self.name` and `self.hot_drink` within our methods, we're telling the methods to look outside the method itself, but stay within the class when trying to find the variables `name` and `hot_drink`. It's a way of determining *scope*: it tells Python to stay within the scope of this class.

This is also why `self` is the first parameter of every method we define inside our class: within a class, every method needs to understand the entire class it's part of. In Python, we must declare `self` as the first parameter of the methods in our class, otherwise the methods are unaware of the other methods and attributes that are also available within the class. So `def speak(self, speech)` has 2 parameters: `self`, by default because all class methods have `self`, and a second one, `speech`, which we use within just this method. Notice that when we refer to the `speech` parameter within the `speak()` method that we don't use `self`? That's because the *scope* of this variable is *inside the method*. By not using `self` here, Python knows when running this code that it should only look for a variable or parameter called `speech` within this method.

Any time you need to reference something *inside* the class but *outside* the method, you'll need `self`.

When we call our method `speak()` later on our `agent_q` object, notice that we skip right over the first `self` argument and only add the second, in this case the text we want to pass as our `speech` argument? Python is a bit picky: it wants us always add `self` as a first parameter when we declare the method, but doesn't want us to add it as an argument: it does this for us automatically every time we call the method.

Creating multiple instances

The real power of classes and objects is the ability to create more than one at a time. Each one is like a copy of the original empty class, and entirely independent of any other objects that we might have previously created using this class as our model.

```
class Agent():
   name = ''
   hot_drink = ''

   def speak(self, speech):
      print(self.name + ' says: "' + speech + '"')

   def drink(self):
      print(self.name + ' drinks a cup of ' + self.hot_drink + '.')

agent_q = Agent()
agent_q.name = 'Agent Q'
agent_q.hot_drink = 'decaf coffee'

agent_m = Agent()
agent_m.name = 'Agent M'
agent_m.hot_drink = 'Earl Grey tea'

agent_q.speak("Hi, I'm Agent Q!")
agent_m.speak("Hi, nice to meet you. I'm Agent M.")
agent_q.drink()
agent_m.drink()
```

In this example, we've created 2 different objects using our class, 1 called `agent_q` and 1 called `agent_m`. We've given each one an appropriate `name` attribute.

Then we get them to speak to each other using the `speak()` method. Notice that each object still knows its own unique `name` attribute when the `speak()` method is run. Setting `agent_m.name = 'Agent M'` doesn't affect how we've set `agent_q.name`. They're modifying different objects.

```
Agent Q says: "Hi, I'm Agent Q!"
Agent M says: "Hi, nice to meet you. I'm Agent M."
Agent Q drinks a cup of decaf coffee.
Agent M drinks a cup of Earl Grey tea.
```

Object constructors

Each time we created a new instance of our class `Agent()` above, we had to first assign its attributes. This is a little bit tedious, so let's update our class and add an *object constructor*. This will let us both instantiate our class and assign our key attributes all on 1 line.

```
class Agent():
    name = ''
    hot_drink = ''

    def __init__(self, name, hot_drink):
        self.name = name
        self.hot_drink = hot_drink

    def speak(self, speech):
        print(self.name + ' says: "' + speech + '"')

    def drink(self):
        print(self.name + ' drinks a cup of ' + self.hot_drink + '.')

agent_q = Agent('Agent Q', 'decaf coffee')
agent_m = Agent('Agent M', 'Earl Grey tea')

agent_q.speak("Hi, I'm Agent Q!")
agent_m.speak("Hi, nice to meet you. I'm Agent M.")
agent_q.drink()
agent_m.drink()
```

We've added a new method to our class, the constructor method. This method has a special name, and if we use it, it should be the very first method we define in our class. As its name "constructor" implies, it helps us "construct" the class more quickly.

The constructor method uses a strange looking name: `__init__()`. That's 2 underscores, followed by 'init', followed by 2 more underscores. This is to make sure this special method never conflicts with any other method you might have in your class.

We give our `__init__()` method 3 parameters here. The first is `self`... because `self` is always the first parameter of a method, and this one follows that rule. Then we also give it `name` and `hot_drink`.

Inside this method, we use these parameters to set the attributes of the object: `self.name = name` and `self.hot_drink = hot_drink`. Here's how we see some of our scope at work: `self.name` refers to the variable *outside* our constructor method but *inside* the class, while `name` in this method refers to the parameter. This is why `self` is important: it differentiates from the `name` inside the method and the `name` inside the class.

Now, when we create an object, we also pass the values we want as attributes into the object as parameters. The first thing that happens when we instantiate a class is that it is *constructed*, and so Python will run the `__init__()` method automatically every time we create a new object, using the arguments we specify at this time.

The output of this new revised code is exactly the same as the output of our previous version, but we've saved some lines and created much nicer looking code.

Modifying object attribute values

Let's add a few more attributes and methods to our class.

```
class Agent():
    name = ''
    hot_drink = ''
    cases_total = 0
    cases_solved = 0

    def __init__(self, name, hot_drink, cases_total, cases_solved):
        self.name = name
        self.hot_drink = hot_drink
        self.cases_total = cases_total
        self.cases_solved = cases_solved

    def speak(self, speech):
        print(self.name + ' says: "' + speech + '"')

    def drink(self):
        print(self.name + ' drinks a cup of ' + self.hot_drink + '.')

    def get_total_cases(self):
        print(self.name + ' has a total of ' + str(self.cases_total) + ' cases.')

    def add_new_case(self, number):
        self.cases_total += number
        if (number > 1):
            print(self.name + ' has been given ' + str(number) + ' new cases.')
        else:
            print(self.name + ' has been given ' + str(number) + ' new case.')

    def get_solved_cases(self):
        print(self.name + ' has solved ' + str(self.cases_solved) + ' cases.')

    def solve_case(self, number):
        self.cases_solved += number
        if (number > 1):
            print(self.name + ' has solved ' + str(number) + ' cases, wow!')
        else:
            print(self.name + ' has solved a case, great job!')

agent_q = Agent('Agent Q', 'decaf coffee', 20, 12)

agent_q.speak("Hi, I'm Agent Q!")

agent_q.get_total_cases()
agent_q.add_new_case(2)
agent_q.get_total_cases()

agent_q.get_solved_cases()
agent_q.solve_case(1)
agent_q.get_solved_cases()
```

In this example, we've added 2 new attributes: `cases_total` and `cases_solved`. We've also updated our constructor to automatically set those attribute values when we construct the object.

We've also added a few new methods to work with these new attributes. We can now request some stats about Agent Q's total and solved cases using `get_total_cases()` and `get_solved_cases()`.

We've also created 2 more methods that let us update attributes of our object. Now when Agent Q gets assigned a new case, or solves a case, we can simply call `add_new_case()` or `solve_case()` on our `agent_q` object, and these attributes will get updated.

Time Remaining: 59 mins Language: Python 2 | Python 3 | C Theme: Dark | Light Status: Editor Terminal Server Server: Reset Log: Download

Steps

Step 1

Create a game that uses 2 classes: `Hero()` and `Monster()`.

The hero:

- Has a weapon that does an amount of damage
- Has an amount of health
- Can attack with their weapon and cause damage
- Can lose health if attacked
- Will die if they have 0 health
- Has 1 healing potion that they can drink to get back 5 health, but the potion can only be used once.

The monster:

- Has an attack strength and can do an amount of damage
- Has an amount of health
- Can roar
- Can attack and cause damage
- Can lose health if attacked
- Will automatically run away if it gets down to 1 health remaining

Step 2

Once you have your classes and objects working, create constructors for each one so you can build them anytime you want with damage strength and starting health points that you choose.

Step 3

Allow your program to accept user input so you can specify the stats of your hero, then have the program randomly assign values to the monster and have them fight, trading attacks back and forth, until the hero dies or the monster runs away.

Editor – file: ~/output.py

```
class Hero():
  def __init__(self, weapon_damage, health, used_potion):
    self.weapon_damage = weapon_damage
    self.health = health
    self.used_potion = used_potion

  def attack(self, monster_health):
    monster_health -= self.weapon_damage
    print(("Hero attacked monster for {0} health, Monster Current Health: {1}").format(self.weapon_damage, mo
    return monster_health

  def use_potion(self):
    self.health += 5
    print(("Hero restored 5 health, Hero Current Health: {0}").format(self.health))
    self.used_potion = True

class Monster():
  def __init__(self, attack_damage, health):
    self.attack_damage = attack_damage
    self.health = health

  def attack(self, hero_health):
    hero_health -= self.attack_damage
    print(("Hero attacked monster for {0} health, Monster Current Health: {1}").format(self.weapon_damage, mon
    return hero_health

  def roar(self):
    print("ROAR!")
```

Save

Terminal – user: QKcyPPGwOk

```
  _____      __            ______          __
 / ___/_ __/ /  ___ ____/ __/ /____ _____/ /_
/ /__/ // / _ \/ -_) __/\ \/ __/ _ `/ __/ __/
\___/\_, /_.__/\__/_/ /___/\__/\_,_/_/  \__/
   /___/                 __  _      __
  / __/__ ___ ___ ___  / /_(_)__ _/ /__
 / _/(_-<(_-</ -_) _ \/ __/ / _ `/ (_-<
/___/___/___/\__/_//_/\__/_/\_,_/_/___/
QKcyPPGwOk@8cf9f6fa3c97:~$
```

Completed
< back to module

Copyright 2020 SANS. Version 1.8.1