Classes in Python

Classes are part of the Object-oriented Programming (OOP) style. We have been writing programs using functions that operate sequentially to solve a problem; that is we look at the problem algorithmically. In contrast, OOP looks at the problem from the perspective of the objects inherent in the problem. Objects are described by their attributes (what makes up the object) and by their methods (what the objects can do)?

A class is an organizational tool that lets us combine data structures with methods that operate on the data. Think of a class as a blueprint or template. Using classes is a two-step process. First, we define the object, creating a template. In this step we define the attributes and the methods. Then, we create instances of the object. Each instance assigns values to the attributes and executes the methods to operate in the real-world.

A car can be used as a simplified example. Attributes for a car include the make, model, color, fuel-level, location and odometer. Methods include gas\_up (changes fuel\_level) and travels (changes fuel\_level, location and odometer). When we create instances of a car, we can keep track of their status by accessing the car object, which is more life-like than accessing lists and arrays.

We can also customize the classes. For example, cars can be divided into subclasses based on their fuel source: electric; gas; diesel; etc. These subclasses all share the original set of attributes and methods but add their own twists; for example, electric cars have battery charge not fuel\_level.

We have been using classes since day 1 (well, maybe day 2). For example,

name = ‘Lee’

assigns the string ‘Lee’ to the variable name. We can find the length of the string by using the len *function*: len(name) → 3. We can make the string all lowercase by using the lower *method*: name.lower() → ‘lee’. We can find how many e’s there are in the string by using the count method: name.count(‘e’) → 2. Notice that len was called a function while lower and count were called methods.

A string is an *instance* of a Python class. A string has data - the characters - and it has methods that are defined to operate on the string (and only on the string). Lower and count (upper, index and format, too) are methods that belong to the string class. In contrast, len is a function that operates on a variety of data types: lists, strings, dictionaries.

Method - defined like a function but operates on a class.

Functions are called by naming the function followed by arguments in parentheses: len(name).

Methods are called by appending the method name to a class instance followed by the arguments in parentheses: name.index(‘e’).

Start with a class for squares. A square consists of several attributes: border\_width, border\_color, fill\_color, length and (locx, locy), where (locx, locy) is the (x, y) coordinate of the upper-left corner of the square. We will also want to compute the area and perimeter of the square. To define a class we use the class statement:

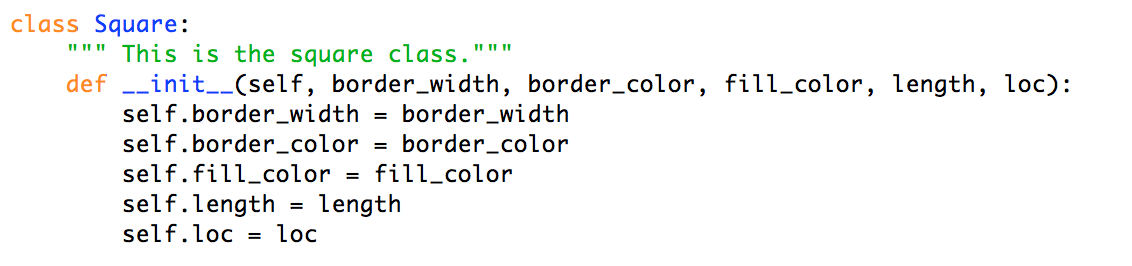
Class Square:

This statement crates the class without specifying any details. To add the attributes, we need to include the \_\_init\_\_() method (those are two underscores before and after init).

Think of turtles. When we write

t = Turtle()

We are creating an instance of the Turtle class (class names are traditionally capitalized). Behind the scenes, Python runs the \_\_init\_\_ method that, in this case, takes no input arguments from us. The \_\_init\_\_ method assigns the initial shape to the turtle, faces it in the initial direction, and sets its initial position (and probably more). The \_\_init\_\_ also creates the *instance* variables to hold these values. If we want to change the shape of a turtle to a circle, we write t.shape (‘circle’), which is how we invoke methods. Only the shape if this turtle instance is affected by calling t.shape().

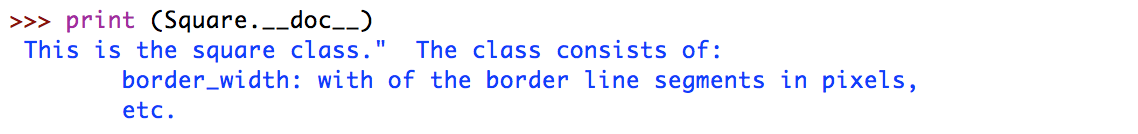


The above code defines the template for a square, it does not define a real square.

The Square class has five instance variables: border\_width, border\_color, fill\_color, length, and loc. These variables are called instance variables because every instance of a square will get their own copy of these variables. Square #1 will not know square #2’s values.

The \_\_init\_\_ method is called the *constructor* for the class.

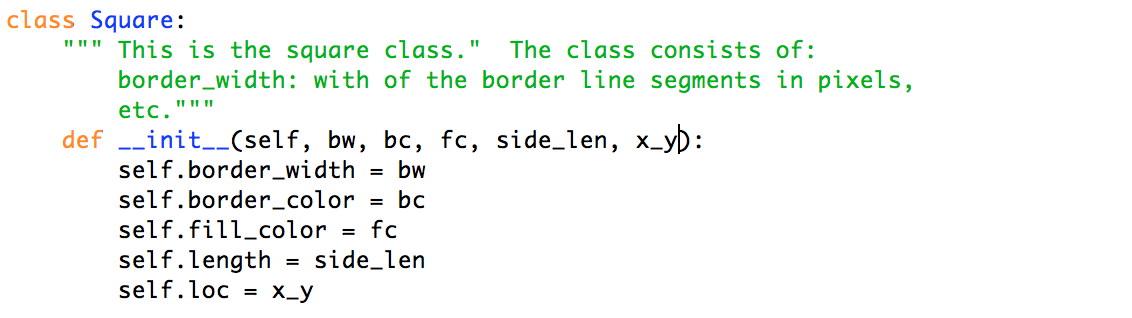
The comment line is part of the class definition. It can be several lines long and must be defined using the triple-quotes (not the #). The comment is saved in the \_\_doc\_\_ instance variable \_\_doc\_\_, which belongs to all classes:



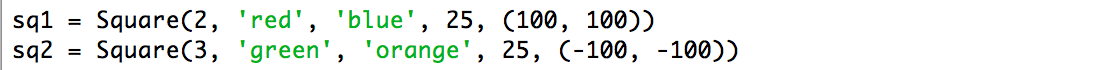
Note: The double-underscore is used as a “hands-off” notice to programmers. Use this variable but do not change it. We can define our own double-underscore instance variables and methods. This style is similar to using all caps for a variable name.

The first parameter of the \_\_init\_\_ is self. self is a reference to the object being created, in this case the Square class.

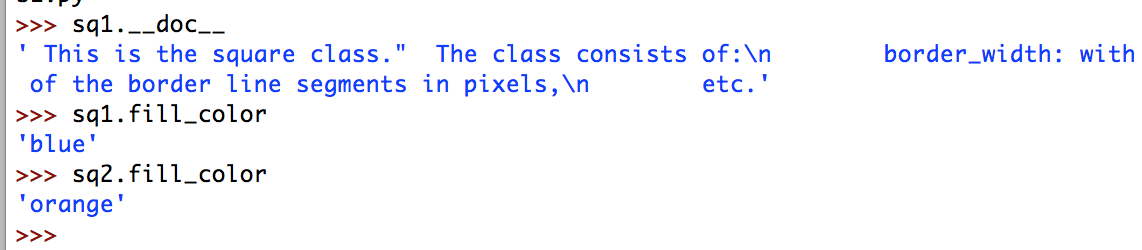
self.border\_width = border\_width : assigns the value passed to the method in border\_width to the instance variable border\_width. Typically, we use the same name in the parameter list as the corresponding instance variable, but we could have written



We will make two instances of a square:



When we run the program nothing seems to happen. However, we can see the effects in the shell window:

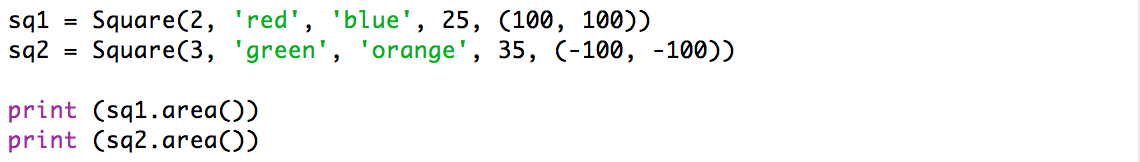


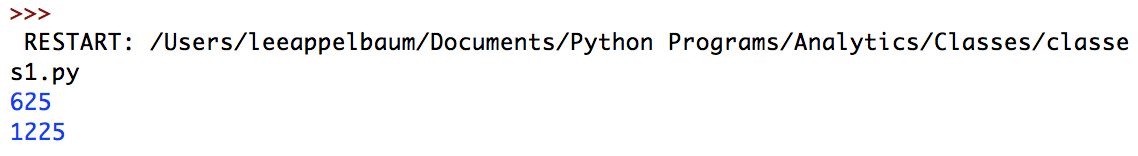
We can compute the area and perimeter using methods.

Create an area method:

The parameter list must start with self.

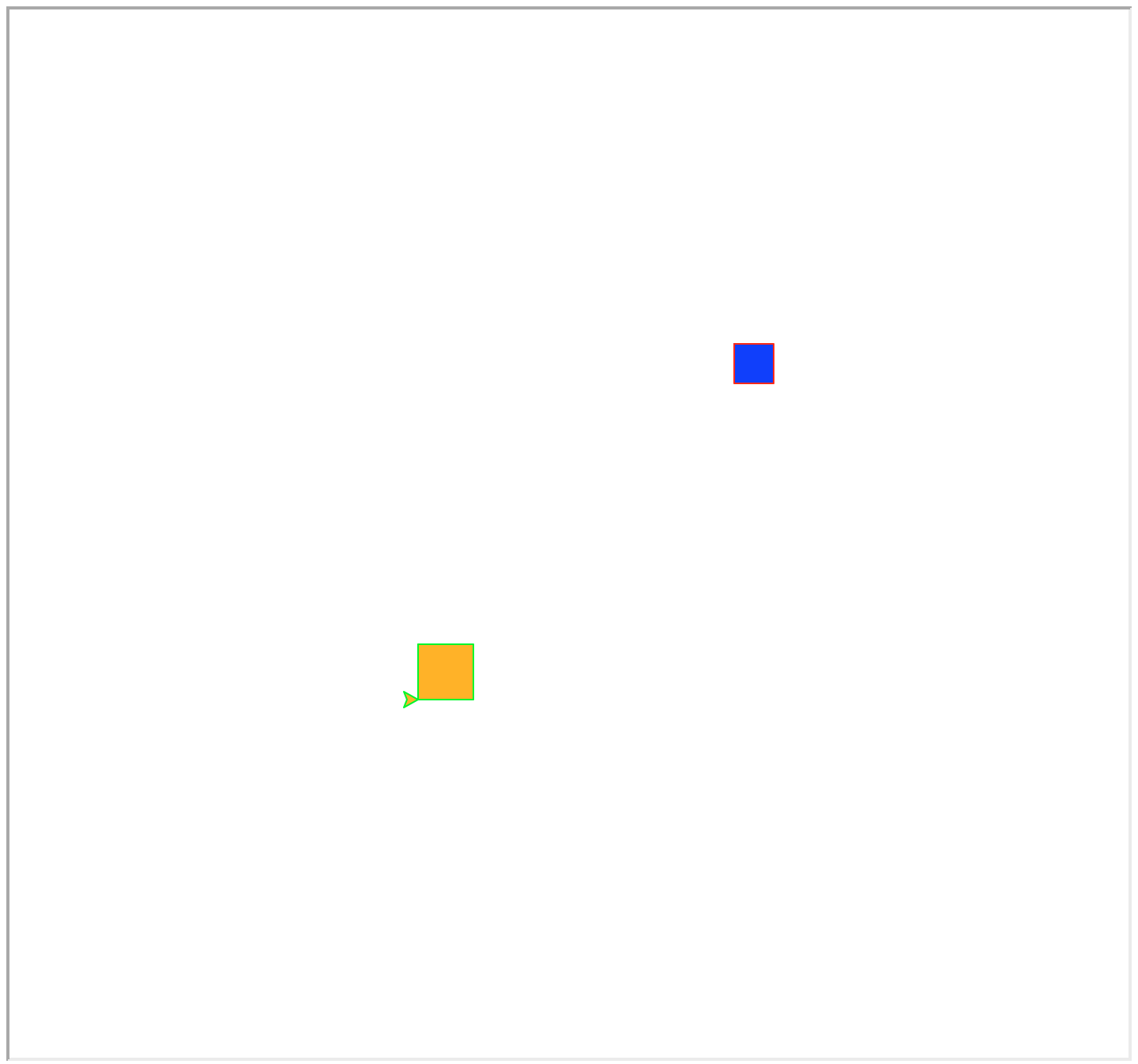
No other parameters are needed since the side length is part of the square definition, the length instance variable.

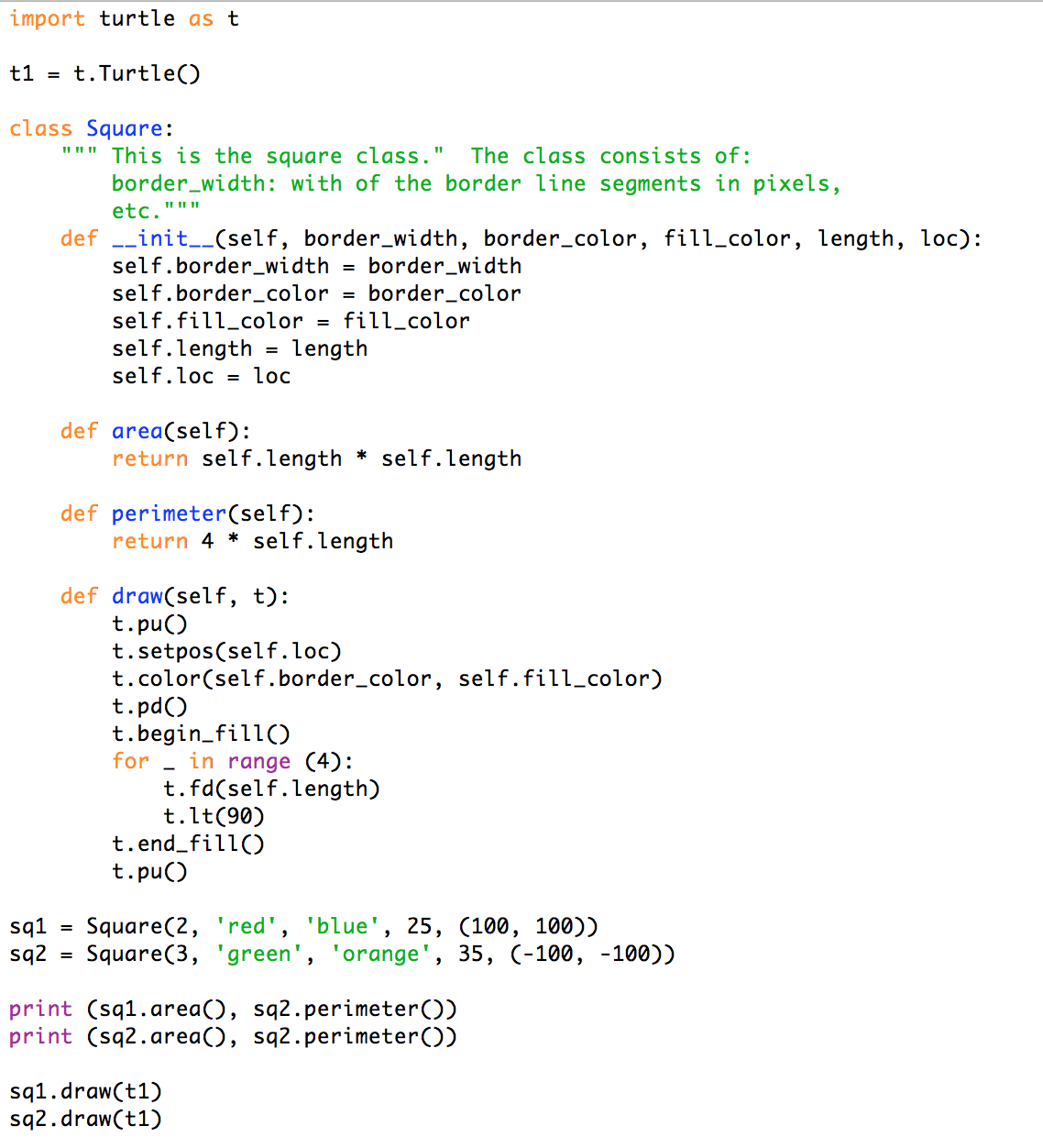




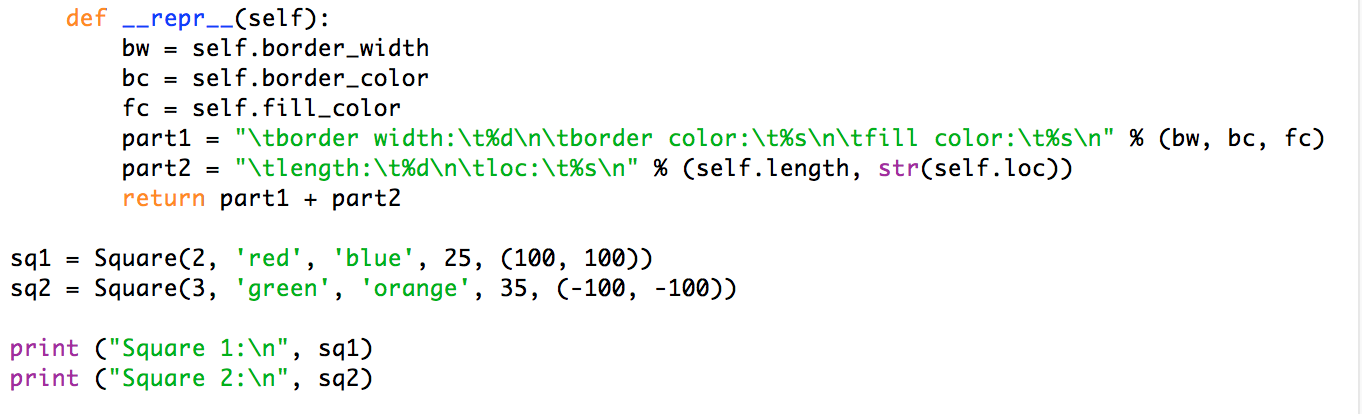
Write a method to compute the perimeter,

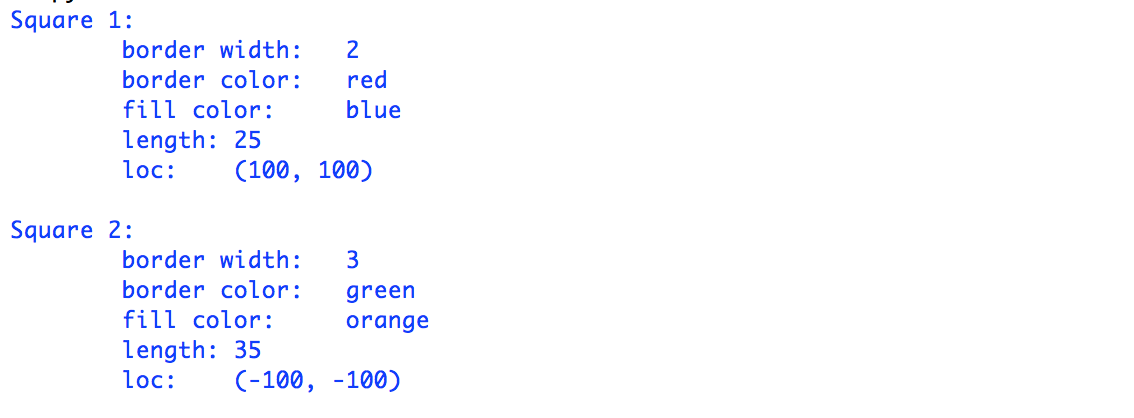
Write a method that uses a turtle to draw the square. Pass the turtle to the method as an argument (the draw method will have two parameters: self and the turtle).





Python includes a built-in method to display the instance variables in a cleanly formatted manner. The method \_\_repr\_\_ can be *overridden* with our own. We redefine \_\_repr\_\_ to return a formatted string to print.



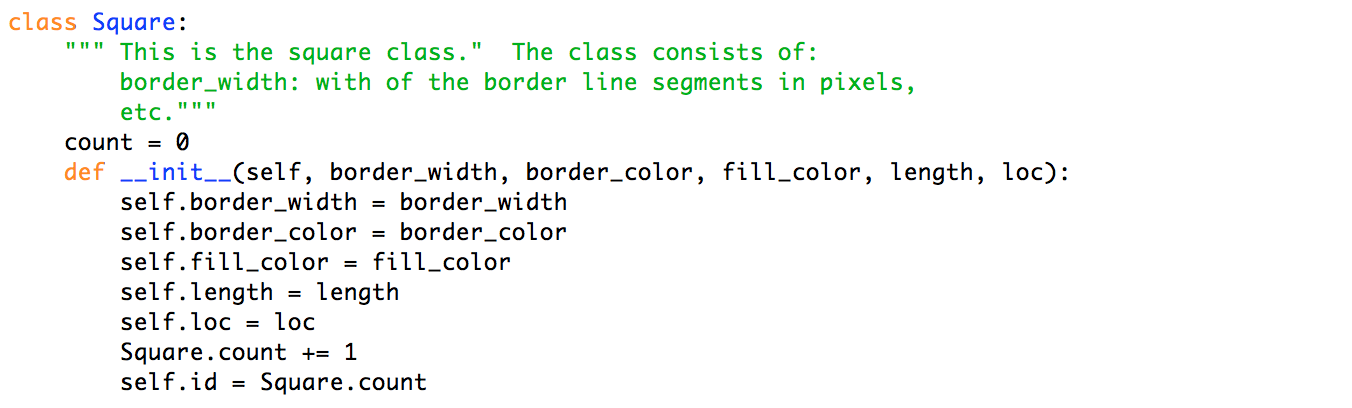


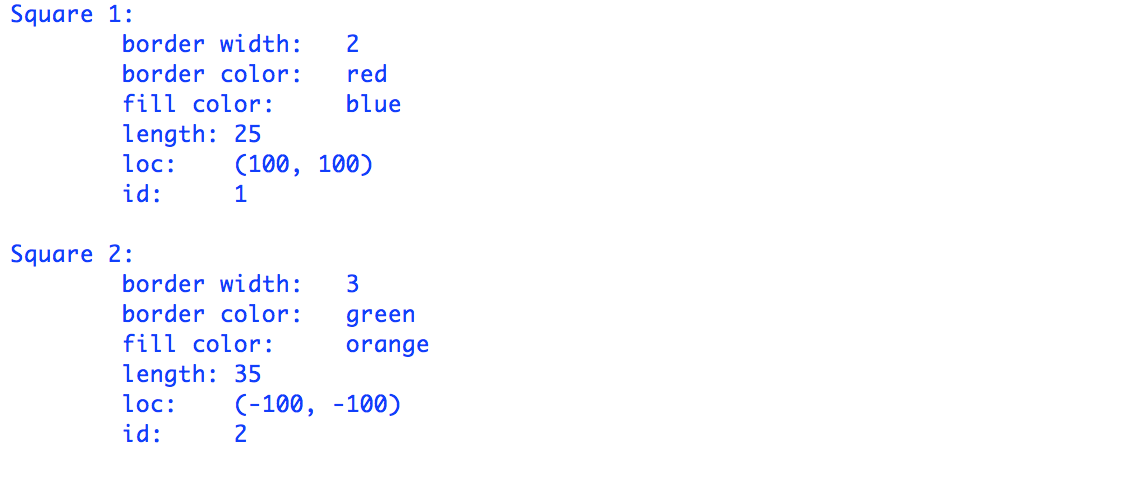
What if we want to assign an id number to each square, starting with 1 and adding 1 for each square instance? We’ll use an instance variable for the id - each square instance has its own private id number - and we will need a count variable that can be accessed by the instances. We can define a variable at the level of the class that all the instances share. We start this variable at 0, add 1 to it in the constructor (the \_\_init\_\_ method), and then assign its value to the id.

We define a class variable without any prefix.

The assignment statement associated with the class variable is executed once, when the class is defined. When the first class instance is created, the variable exists and is initialized per the assignment.

To access a *class variable*,we prepend the name of the class to the variable name.





We have just demystified the Turtle!

Turtle is a class. It’s constructor method (\_\_init\_\_()) has no mandatory parameters, other than self, which we do not see.

t = Turtle() creates an instance of the turtle class and calls the constructor.

pd(), pu(), fd(), shape(), … are al methods of the Turtle class. Since methods are attached to a class, we call them by appending them to a Turtle instance: t.pd(), t.pu(), t.fd(), …

When we write print(t) we get the result of the \_\_repr\_\_ method written by the Turtle programmers.

When we write print(t.\_\_doc\_\_) we get the comment written by the Turtle programmers right after the “class Turtle” line (in their code - we do not see it).

We will see how to make subclasses in the next lesson. We can create a subclass of Turtle and write our own version of \_\_repr\_\_ to include the information we want to present. Rewriting a method is referred to as overriding the method.

--- may be all for day 1 ---

What if we want to make triangles, pentagons, etc.? Let’s stay with regular polygons for now. We can define a class for each of these polygons, but we would be repeating a lot of information. In fact, the only difference between the polygons is the number of sides (also the angles). The solution is to create a regular polygon class and have the specific polygons inherit from the general one, adding their own customized data.

Following is work in progress.

We will use 2-D shapes as the setting for introducing classes. For us, 2-D shape consists of polygons and circles. A shape by itself has no structure; you need to specify what kind of shape before we can picture it. However, all shapes have a couple of attributes in common; namely, they all have a border width, border color and a fill color. All specific shapes (rectangles, triangles, etc.) inherit these properties and add some of their own.

A method is defined just like a function except that it is indented under the class statement AND is always includes the parameter self.

