**Set and get methods**

The last part of the video mentioned that accessing attributes in Python can be somewhat different than in other programming languages like Java and C++. This section goes into further detail.

The Shirt class has a method to change the price of the shirt: shirt\_one.change\_price(20). In Python, you can also change the values of an attribute with the following syntax:

shirt\_one.price = 10

shirt\_one.price = 20

shirt\_one.color = 'red'

shirt\_one.size = 'M'

shirt\_one.style = 'long\_sleeve'

This code accesses and changes the price, color, size, and style attributes directly. Accessing attributes directly would be frowned upon in many other languages, **but not in Python**. Instead, the general object-oriented programming convention is to use methods to access attributes or change attribute values. These methods are called set and get methods or setter and getter methods.

A get method is for obtaining an attribute value. A set method is for changing an attribute value. If you were writing a Shirt class, you could use the following code:

class Shirt:

def \_\_init\_\_(self, shirt\_color, shirt\_size, shirt\_style, shirt\_price):

self.\_price = shirt\_price

def get\_price(self):

return self.\_price

def set\_price(self, new\_price):

self.\_price = new\_price

Instantiating and using an object might look like the following code:

shirt\_one = Shirt('yellow', 'M', 'long-sleeve', 15)

print(shirt\_one.get\_price())

shirt\_one.set\_price(10)

In the class definition, the underscore in front of price is a somewhat controversial Python convention. In other languages like C++ or Java, price could be explicitly labeled as a private variable. This would prohibit an object from accessing the price attribute directly like shirt\_one.\_price = 15. Unlike other languages, Python does not distinguish between private and public variables. Therefore, there is some controversy about using the underscore convention as well as get and set methods in Python. Why use get and set methods in Python when Python wasn't designed to use them?

At the same time, you'll find that some Python programmers develop object-oriented programs using get and set methods anyway. Following the Python convention, the underscore in front of price is to let a programmer know that price should only be accessed with get and set methods rather than accessing price directly with shirt\_one.\_price. However, a programmer could still access \_price directly because there is nothing in the Python language to prevent the direct access.

To reiterate, a programmer could technically still do something like shirt\_one.\_price = 10, and the code would work. But accessing price directly, in this case, would not be following the intent of how the Shirt class was designed.

One of the benefits of set and get methods is that, as previously mentioned in the course, you can hide the implementation from your user. Perhaps, originally, a variable was coded as a list and later became a dictionary. With set and get methods, you could easily change how that variable gets accessed. Without set and get methods, you'd have to go to every place in the code that accessed the variable directly and change the code.

You can read more about get and set methods in Python on this [Python Tutorial site](https://www.python-course.eu/python3_properties.php).

**Attributes**

There are some drawbacks to accessing attributes directly versus writing a method for accessing attributes.

In terms of object-oriented programming, the rules in Python are a bit looser than in other programming languages. As previously mentioned, in some languages, like C++, you can explicitly state whether or not an object should be allowed to change or access an attribute's values directly. Python does not have this option.

Why might it be better to change a value with a method instead of directly? Changing values via a method gives you more flexibility in the long-term. What if the units of measurement change, like if the store was originally meant to work in US dollars and now has to handle Euros? Here's an example:

**Example: Dollars versus Euros**

If you've changed attribute values directly, you'll have to go through your code and find all the places where US dollars were used, such as in the following:

shirt\_one.price = 10 # US dollars

Then, you'll have to manually change them to Euros.

shirt\_one.price = 8 # Euros

If you had used a method, then you would only have to change the method to convert from dollars to Euros.

def change\_price(self, new\_price):

self.price = new\_price \* 0.81 # convert dollars to Euros

shirt\_one.change\_price(10)

For the purposes of this introduction to object-oriented programming, you don't need to worry about updating attributes directly versus with a method; however, if you decide to further your study of object-oriented programming, especially in another language such as C++ or Java, you'll have to take this into consideration.

**Modularized code**

Thus far in the lesson, all of the code has been in Jupyter Notebooks. For example, in the previous exercise, a code cell loaded the Shirt class, which gave you access to the shirt class throughout the rest of the notebook.

If you were developing a software program, you would want to modularize this code. You would put the Shirt class into its own Python script, which you might call shirt.py. In another Python script, you would import the Shirt class with a line like from shirt import Shirt.

For now, as you get used to OOP syntax, you'll be completing exercises in Jupyter Notebooks. Midway through the lesson, you'll modularize object-oriented code into separate files.

;