**Object-Oriented Python**

Contents

[1. Introduction 1](#_Toc112243119)

[2. Classes and Objects 3](#_Toc112243120)

[3. Defining a Class 5](#_Toc112243121)

[4. Instantiating a Class 6](#_Toc112243122)

[5. Creating Methods 8](#_Toc112243123)

[6. Understanding "Self'" 10](#_Toc112243124)

[7. Creating a Method that Accepts an Argument 13](#_Toc112243125)

[8. Attributes and the Init Method 15](#_Toc112243126)

[9. Creating an Append Method 19](#_Toc112243127)

[10. Creating and Updating an Attribute 21](#_Toc112243128)

# Introduction

**Procedural programming** involves writing code in a number of sequential steps; sometimes we combine these steps into commands called **functions.**

In this lesson, we're going to learn about a new system: **object-oriented programming (OOP).** Rather than designing code around sequential steps, we will design it around objects. For now, consider objects to be closely related to variables. We'll learn a more formal definition on the next screen.

Here are a few takeaways you can expect by the end of the lesson:

- How object-oriented programming relates to the data science workflow

- Key concepts, including classes, instances, attributes, and methods

- How to create your own class

When working with data, it's much more common to use a style that is closer to procedural programming style than OOP, but it's very important to understand how OOP works, because Python is an object-oriented language.

This means almost everything in Python is actually an object; when you're working with Python, you are creating and manipulating objects. As you continue to learn to work with data in Python, you'll encounter objects everywhere:

* NumPy and pandas — the two libraries essential to working with data in Python: both define a number of their own object types
* Matplotlib — which you use to create data visualizations: uses object types to define the charts you create
* Scikit-learn — which you use to create machine learning models: uses object types to represent the models you train and use to make predictions

While it's much less common for data scientists and data analysts to define new types of objects, we'll be using objects all the time

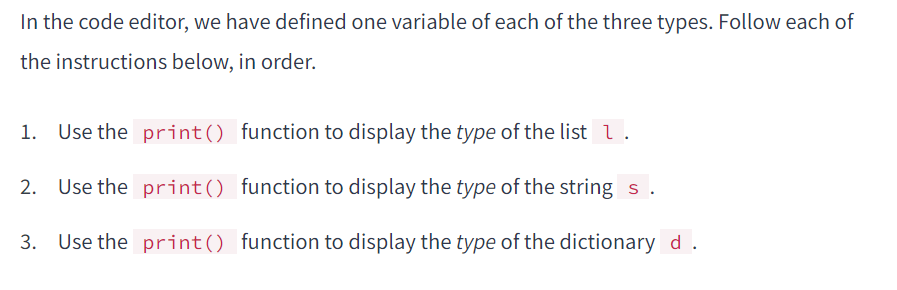
In OOP, objects have types, but instead of "type" we use the word **class**. Here are the correct names for each of these classes:

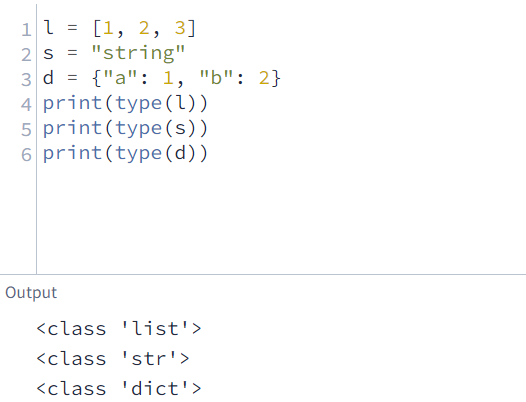
* String class
* List class
* Dictionary class

In everyday English, the word class refers to a group of similar things. In OOP, we use the word similarly — a class is a type of object.

When talking about programming, we often use the words "type" and "class" interchangeably, but "class" is more formally about objects. Throughout this lesson, we'll be using "class" as we learn about OOP.

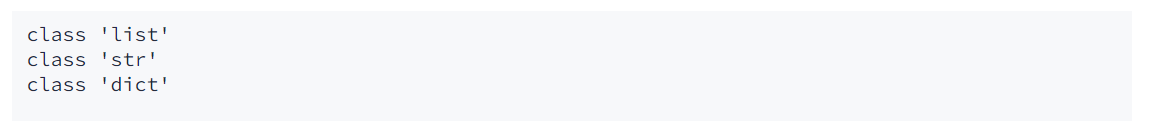
Let's do a quick exercise using the [type() function](https://docs.python.org/3.7/library/functions.html#type) to return variable types.



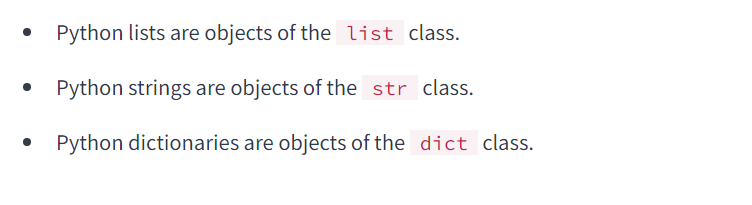


# Classes and Objects

When we used the type() function on the previous screen, it returned values labeled "class":



This demonstrates how we can use "type" and "class" interchangeably. This reveals that we've been using classes for some time already:



In this lesson, we're going to learn how classes work by creating one of our own. We're going to create a simple class called MyList and recreate some of the basic functionality of the Python list class.

As we mentioned earlier, it's less common for data scientists and data analysts to define new types of objects, but understanding how objects work will be extremely valuable to you as you continue to extend your Python knowledge and work more with objects.

Before we begin, let's learn more about objects and classes. Earlier, we said that an object is similar to a variable. As with "class" and "type," we use the two terms interchangeably. We'll learn the subtle difference between objects and variables later, but for now, let's look at the relationship between objects and classes:

An **object** is an entity that stores data.

An object's **class** defines specific properties that objects of that class will have.

One way to understand the difference between a class and an object in Python is by comparing them to real-world objects. We'll compare Python string objects to Tesla electric cars.

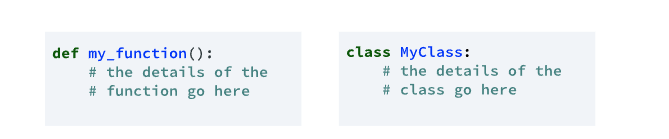


There are hundreds of thousands of Tesla cars around the world. Each car is similar in that it is a Tesla — it's not a Ford or Toyota — but at the same time, it is not necessarily identical to other Teslas. We would say that each of the cars are objects that belong to the Tesla class.

Tesla has a design for making their cars. The design defines what the car is, what it does, and how it does it — everything that makes the car unique. That said, the design isn't a car; it's just all the information necessary to create the car. Similarly, in Python, we have designs for classes. These designs are **class definitions.**

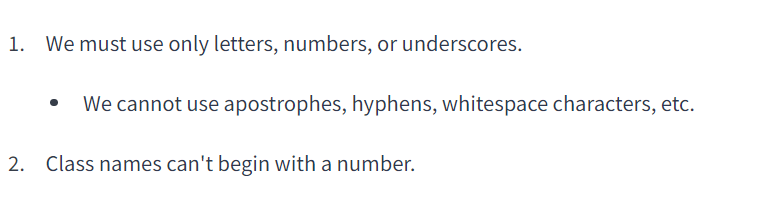
# Defining a Class

We define a class similarly to how we define a function:



Notice that the class definition doesn't have parenthesis (). This is optional for classes, and the convention is to not use them. Similarly, we indent the body of our class like a function's body.

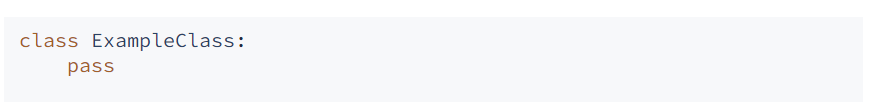
The rules for naming classes are the same as naming functions and variables:



That said, there is a convention for variables and functions in Python called **Snake Case**, which uses all lowercase letters and underscores: like\_this. With classes, the convention is to use **Camel Case**, where no underscores appear between words, and we capitalize the first letter of each word: LikeThis

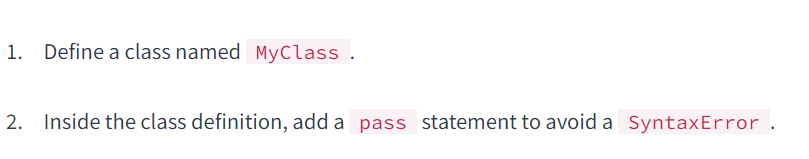
If we try to run either of the code examples in the diagram at the top of this screen, we will get a SyntaxError because Python doesn't let us define classes or functions when they are empty:

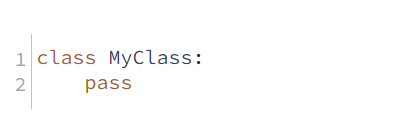
We can use the [pass statement](https://docs.python.org/3/reference/simple_stmts.html#pass) to avoid this error. The pass statement doesn't do anything, but it lets us define an empty code block. Let's see how that looks:



The pass statement is useful if you're building something complex and you want to create a placeholder for a function that you will build out later without causing any error. We'll use the pass statement on this screen to define an empty class without causing an error.

Let's use what we've learned to create the very first class!

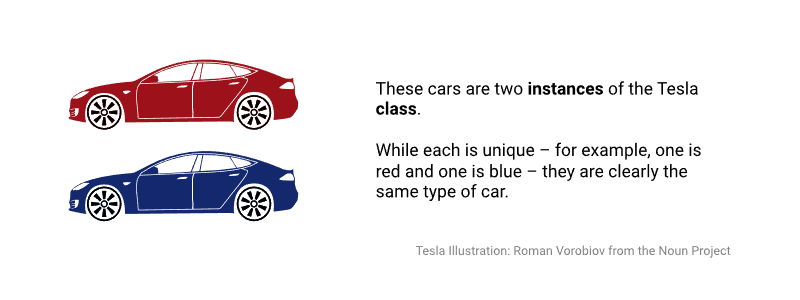




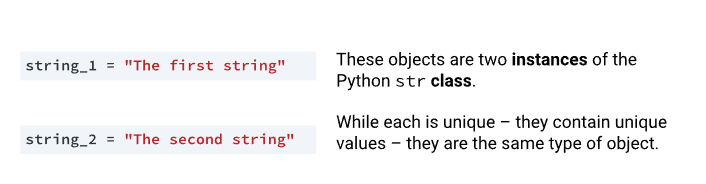
# Instantiating a Class

Earlier, we compared objects to Tesla cars to help us understand the distinction between a class and an object. Let's extend that analogy to help us understand more about classes.

In OOP, we use **instance** to describe each different object. Let's look at an example:



We can say the same of Python strings. We might create two Python strings, and they can hold different values, but they work the same way:



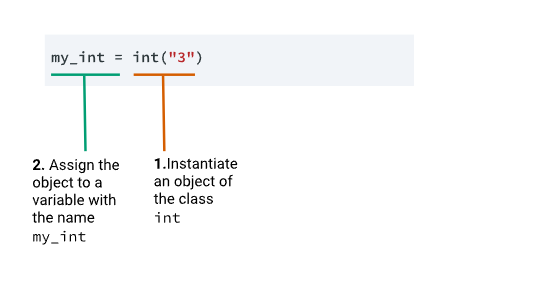
Once we have defined our class, we can create an object of that class, which we call **instantiation**. If you create an object of a particular class, the technical phrase for what you did is to "**Instantiate** an object of that class." Let's learn how to instantiate an instance of our new class:



That single line of our code actually did two things:

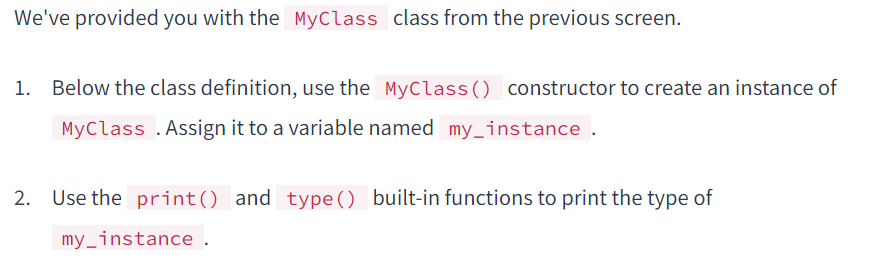
* Instantiated an object of the class ExampleClass
* Assigned that instance to the variable named my\_class\_instance

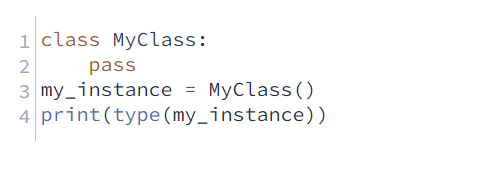
To illustrate this more clearly, let's look at an example using Python's built-in integer class. Let's look at a simple example using the syntax int() to convert numeric values stored as strings to integers. We'll break down the syntax into parts, which we'll read right-to-left:



The syntax to the right of the assignment operator (=) instantiates the object, and the assignment operator and variable name create the variable. This helps us understand some of the subtle differences between an object and a variable.

Keep in mind that in casual usage, we can use "object" and "variable"' interchangeably. The distinction is usually only important if you're talking about OOP concepts like classes.

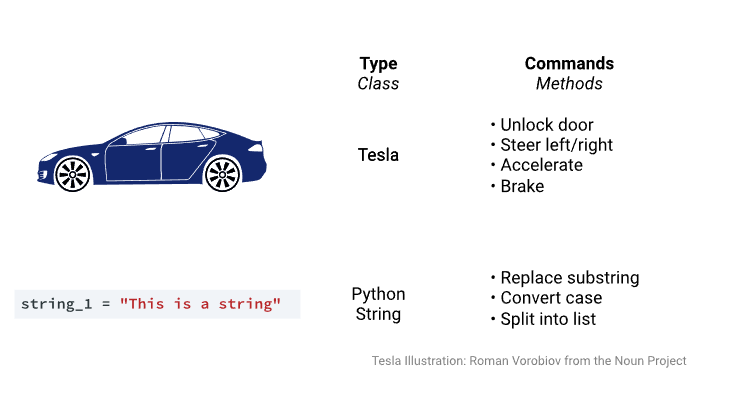




# Creating Methods

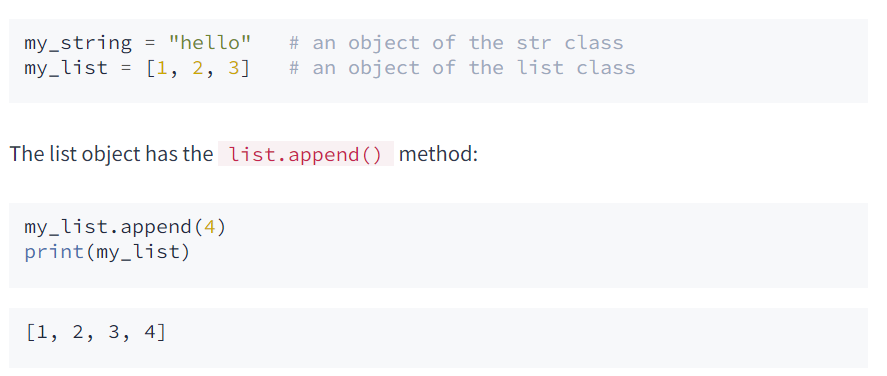
 In order to make our class do something, we need to define some **methods**. Methods allow objects to perform actions.

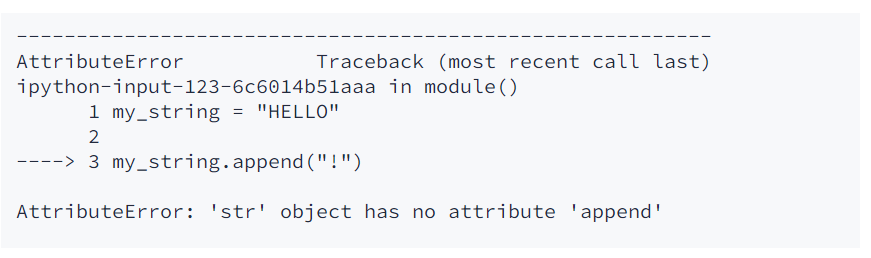
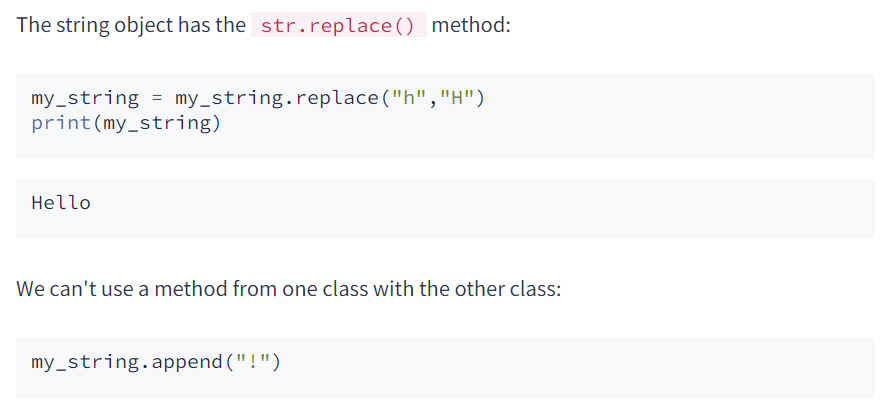
Relating back to our Tesla metaphor, an object of the Tesla "class" can do things like "unlock" and "accelerate." Similarly, Python strings have methods that can replace substrings, convert the case of the object, and more:



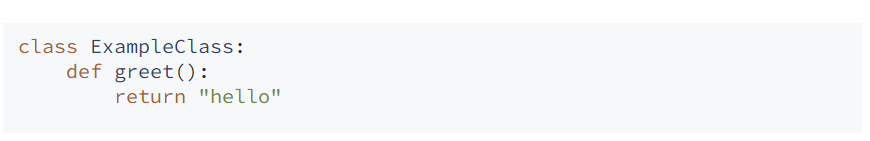
You can think of methods like special functions that belong to a particular class. This is why we call the replace method str.replace()— because the method belongs to the str class.

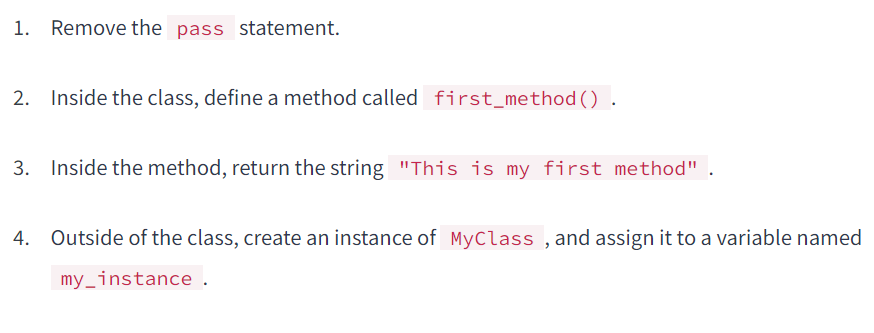
While we can use a function with any object, each class has its own set of methods. Let's look at an example using some of Python's built-in classes:

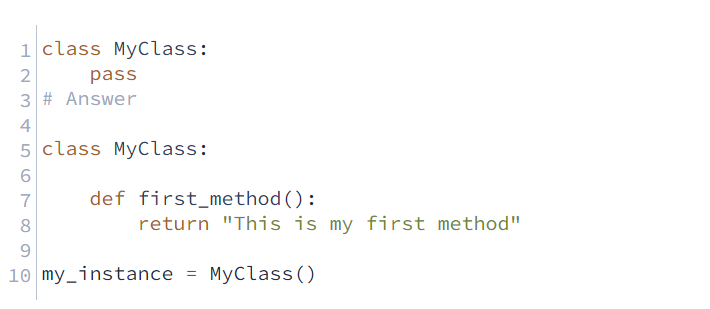




The syntax for creating a method is almost identical to creating a function, except we indent it within our class definition. This is how we would define a simple method:



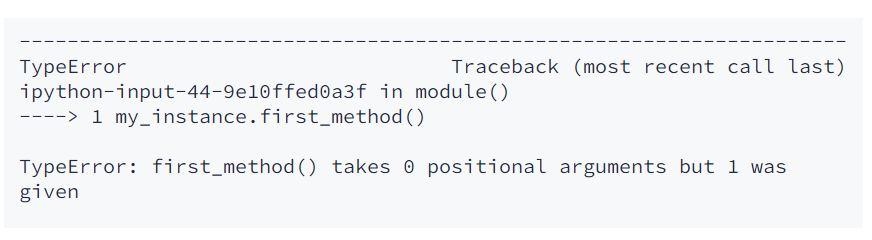




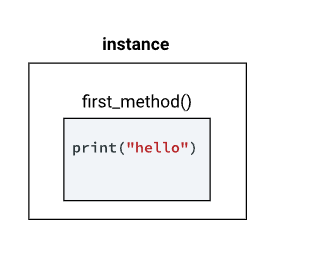
# Understanding "Self'"

On the previous screen, we defined a class with a simple method, then created an instance named my\_instance of that class:

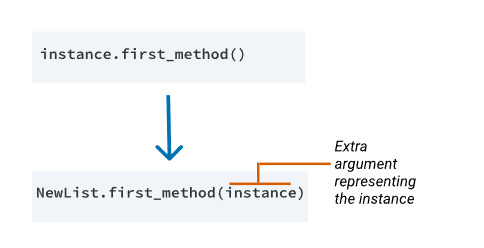




This error is confusing. It says that one argument was given to first\_method(), but when we called the method, we didn't provide any arguments. It seems like there is a "phantom" argument being inserted somewhere. To understand what's happening, let's look at what happens when we call a method. We'll start by looking at our my\_instance object containing a single method:



When we call the first\_method() method belonging to the my\_instance object, Python interprets that syntax and adds an argument representing the instance on which we're calling the method:

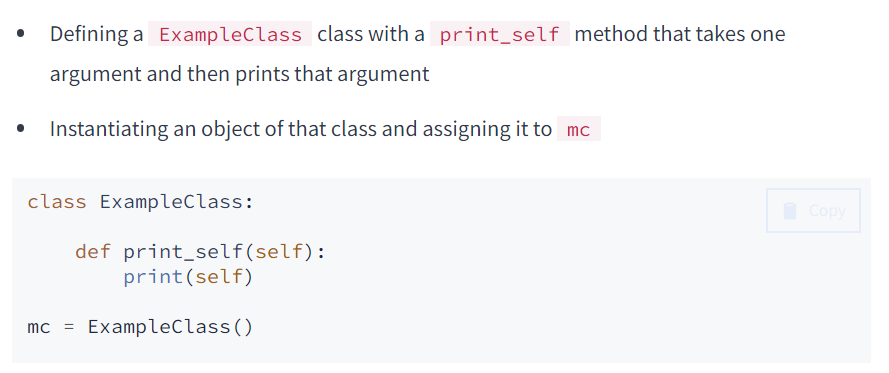


We can verify that this is the case by checking it with Python's built-in str type. We'll use str.title() to convert a string to title case

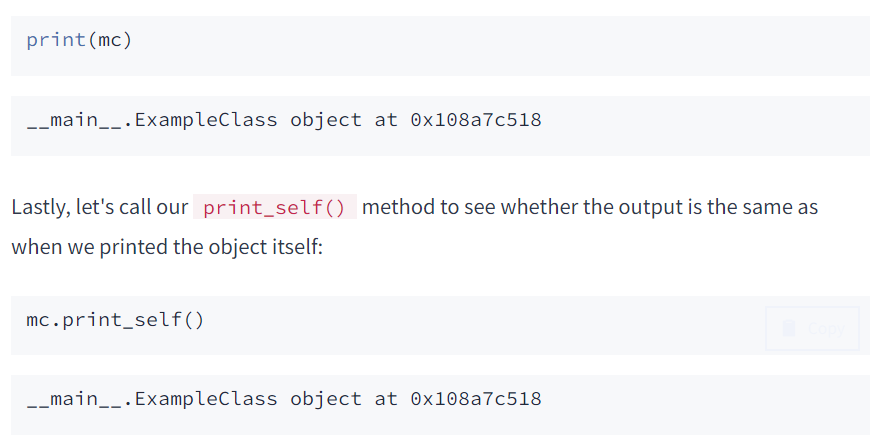


The extra argument that Python has added, which is the instance itself, is causing our error. You might be wondering if we can prove that the extra argument is the object itself? Let's see if we can:

We'll start by doing the following:



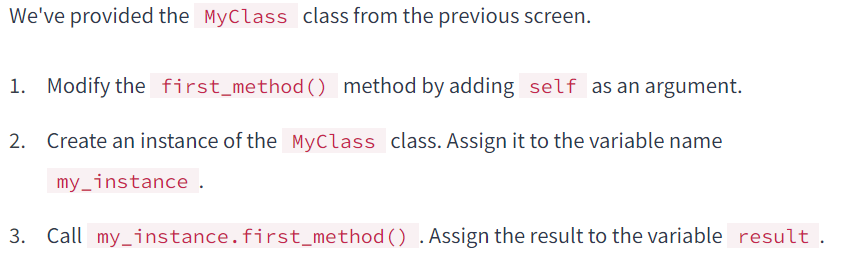
Next, let's print the mc object so we can understand what the object itself looks like when it prints:

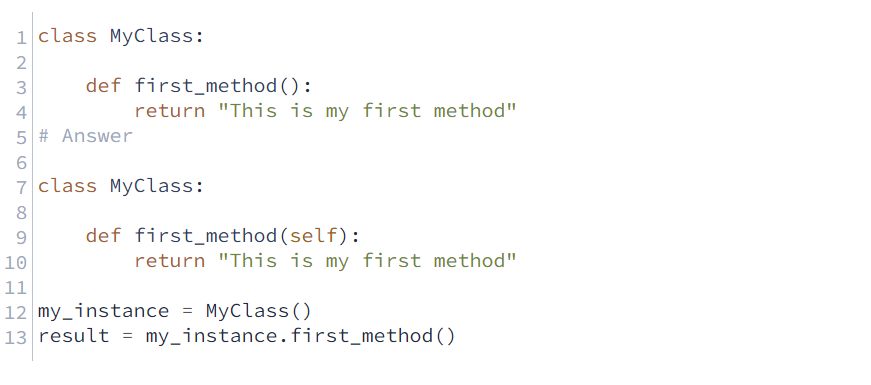


The same output displayed both when we printed the object using the syntax print(mc) and when we printed the object inside the method using print\_self() — which proves that this "phantom" argument is the object itself!

Technically, we can give this first argument — which passes to every method — any parameter name we like. However, the convention is to call the parameter self. This is an important convention because, without it, class definitions can become confusing.

Let's modify the class we created on the previous screen by adding self as an argument to our method. Then, let's call the method to make sure it runs without error.





# Creating a Method that Accepts an Argument

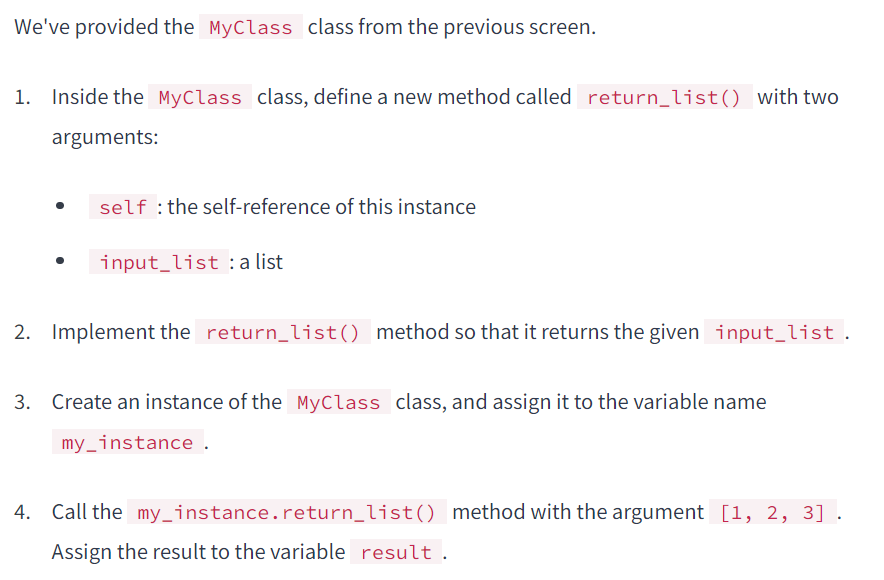
On the previous screen, we learned the following:

* A "phantom" argument passes methods when we call them.
* The "phantom" argument is actually the object itself.
* We need to include that in our method definition.
* The convention is to call the "phantom" argument self

The method we worked with on the previous two screens didn't accept any arguments other than the self argument. As with functions, we often call methods with one or more arguments so that the method can use or modify that argument.

Let's create a method that accepts a string argument and then returns that string. The first argument will always be the object itself, so we'll specify self as the first argument and the string as our second argument:







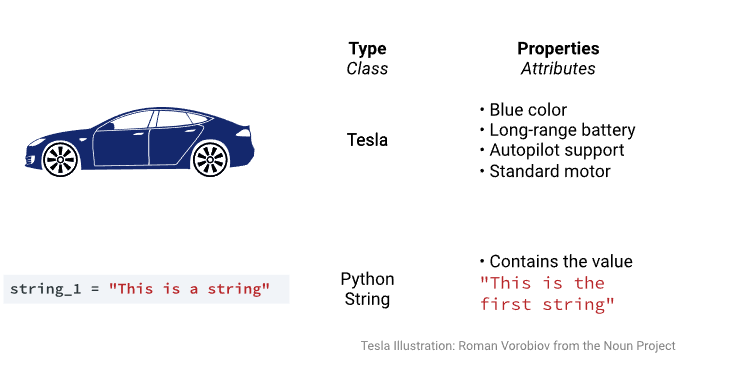
# Attributes and the Init Method

The example we used on the previous screen — a method that takes input and returns output without interacting with the object — isn't a technique we use often.

After all, we could do the same thing with a function without the hassle of defining a class and method. We used this example so you could practice creating a simple class with what you've learned so far.

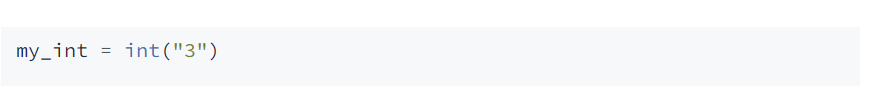
The power of objects is in their ability to store data using **attributes**

Relating back to our Tesla metaphor, an object of the Tesla "class" has attributes like their color, battery, and motor. Similarly, Python strings have attributes — the data stored inside the string:



You can think of attributes like special variables that belong to a particular class. Attributes let us store specific values about each instance of our class.

When we instantiate an object, we usually specify the data that we want to store inside that object. Let's look at an example of instantiating an int object:

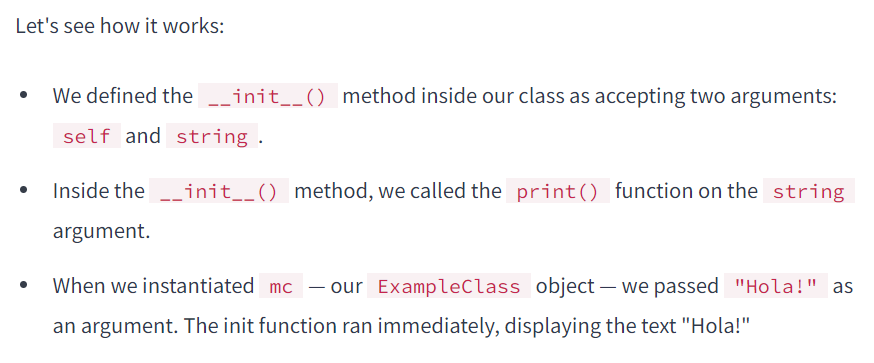


When we used int(), we provided the argument "3", which was converted and stored inside the object. We define any arguments we provide at instantiation using the **init method.**

The init method — also called a **constructor** — is a special method that runs when we create an instance so we can perform any tasks to set up the instance.

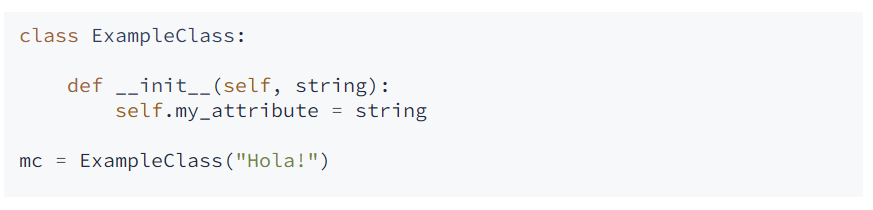
The init method has a special name that starts and ends with two underscores: \_\_init\_\_(). Let's look at an example:



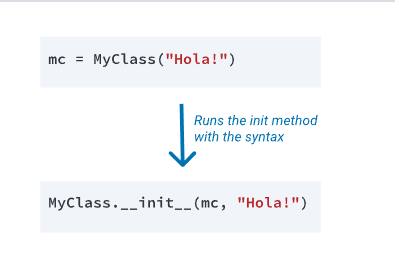


It's unusual to use print() inside an init method, but it helps us understand that the method has access to any arguments passed when we instantiate an object.

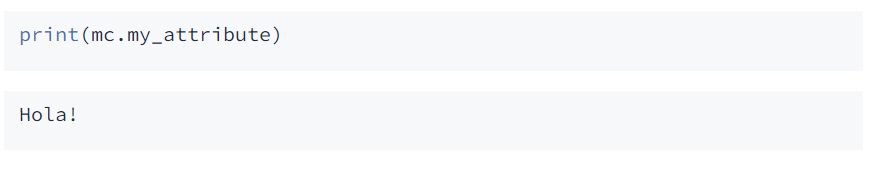
The init method's most common usage is to store data as an attribute:



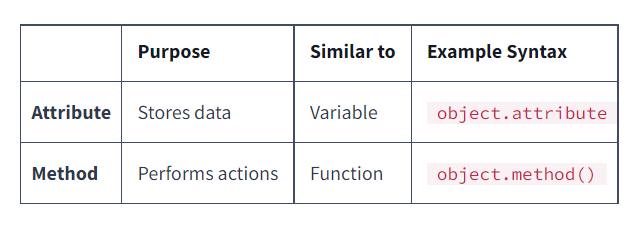
When we instantiate our new object, Python calls the init method, passing in the object:

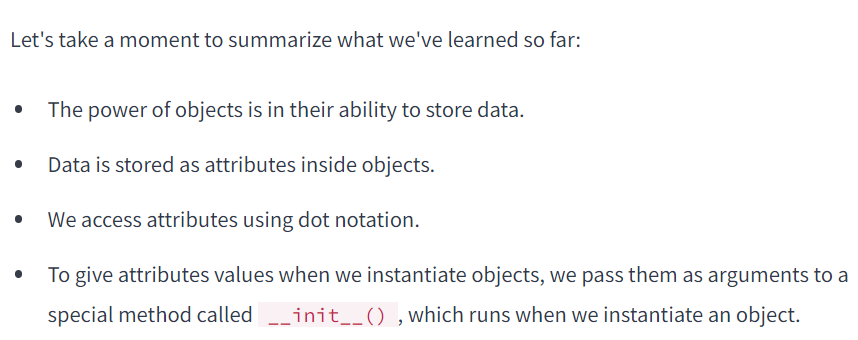


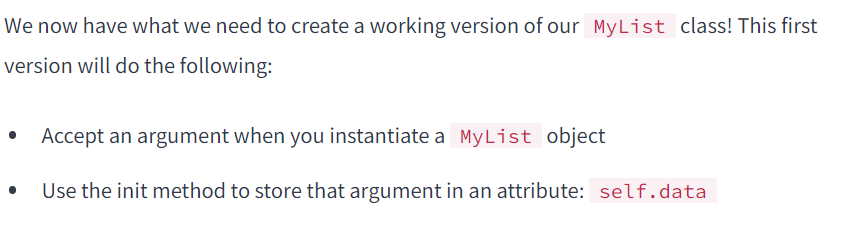
Our code didn't result in any output, but now we have stored "Hola" in the attribute my\_attribute inside our object. Like methods, we access attributes using dot notation, but attributes don't have parentheses like methods do. Let's use dot notation to access the attribute:

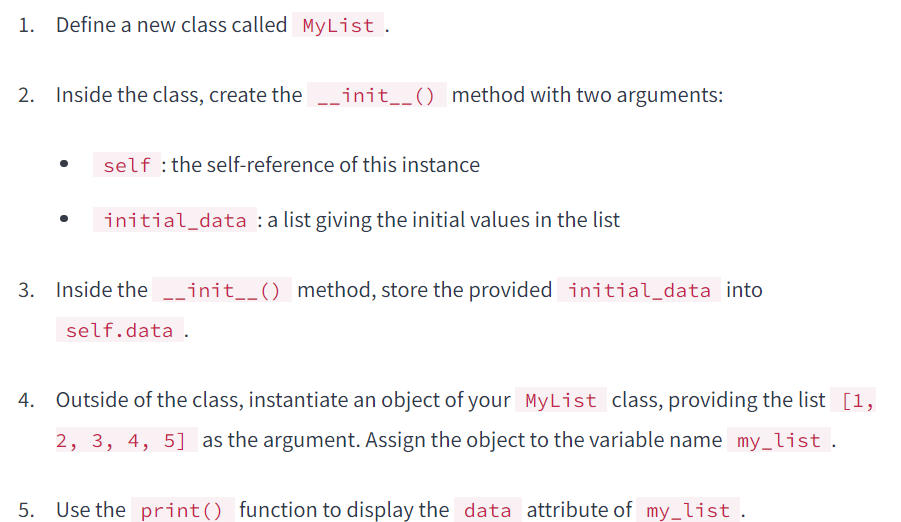


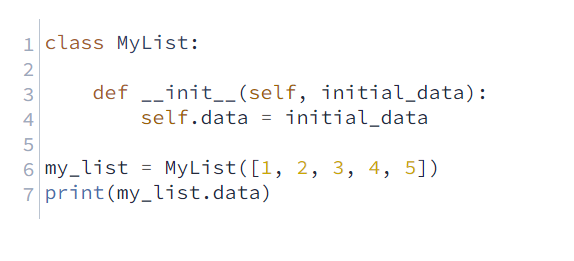
The table below summarizes some of the differences between attributes and methods:







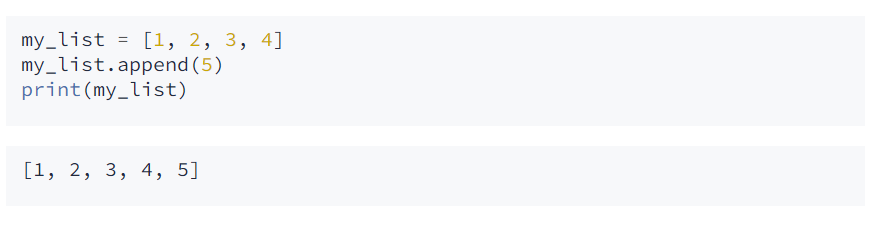


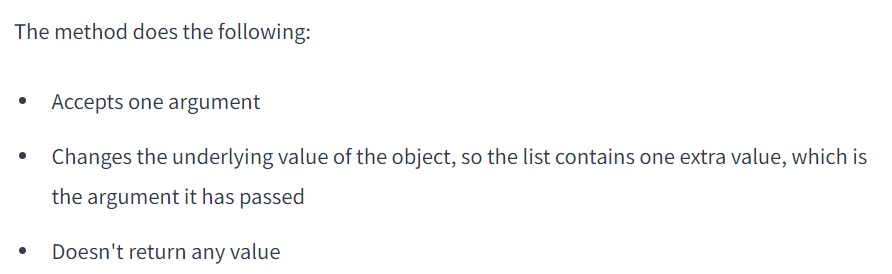


# Creating an Append Method

It's time to create a method to transform the data stored in our MyList objects. We'll be recreating the functionality of the list.append() from the built-in Python list class

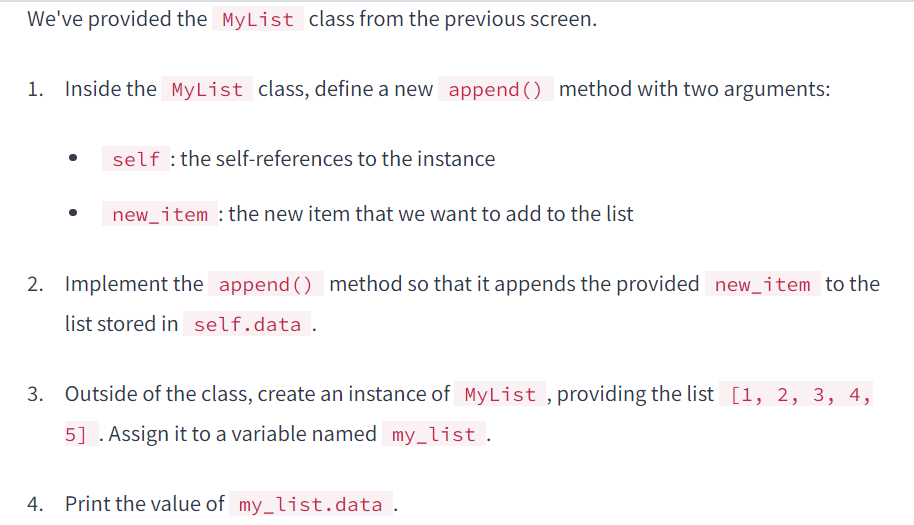
Let's start by looking at an example of list.append() in action:

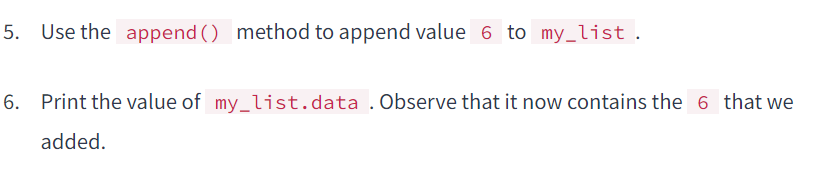


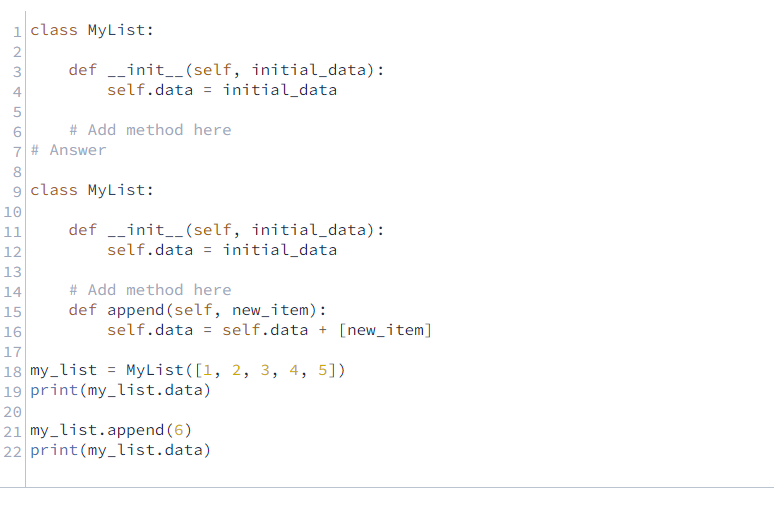
In order to create this method, we need a way to add one extra item to a list. One straightforward way is to add brackets around the second item, making it a list with a single item, then use the + operator to join those two lists:



We now have everything we need to create the MyList.append() method. Remember that to access our attribute from within the method, we need to use self.data, just like we did with the \_\_init\_\_() method

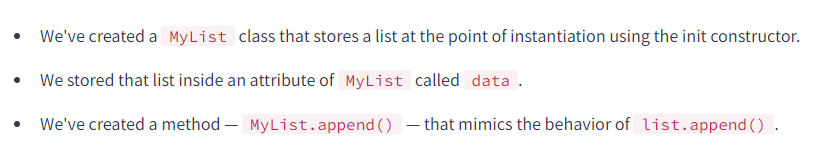






# Creating and Updating an Attribute

Let's summarize the work we've done so far

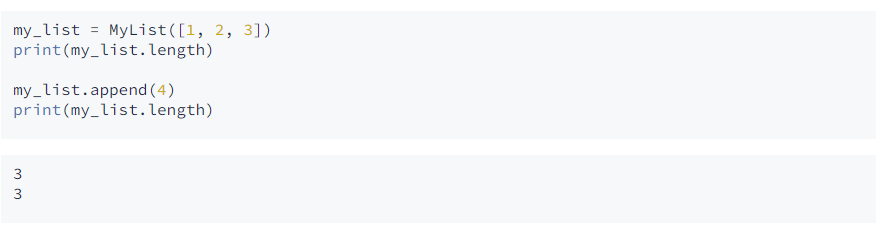


Right now, each behavior we've created for our MyList class is also something a regular Python list does. Now we're going to create some new functionality: a new attribute.

When we want to find the length of a list, we use the len() function. What if we created a new attribute, MyList.length, which stores the length of our list at all times? We can achieve this by adding some to the \_\_init\_\_() method:



Let's look at what happens when we use the MyList.length attribute as defined above:



Because the code we added that defined the length attribute was only in the init method, if the list becomes longer using the append() method, our length attribute is no longer accurate.

To address this, we need to run the code that calculates the length after any operation that modifies the data, which, in our case, is just the append() method. More precisely, we need to increment the length of the list each time we append a value to it.

Let's modify the append() method so that it updates the length of the list when we use it.

