Introduction to Classes

by Sophia

WHAT'S COVERED

In this lesson, we will explore the basics of object-oriented programming and define its structure, which includes classes. Specifically, this lesson covers:

1. Recap and OOP

Before we discuss object-oriented programming and classes, let's review a few things that we have learned in this course up to this point.

In Units 1 and 2, we learned the four basic programming structures for writing code:

1. Sequential code - a fancy way to state lines of code that do something in sequence.
2. Conditional code - using if, else, and elif statements to check for certain conditions and act on those conditions when found (or not found).
3. Repetitive code - Using `for` and `while` loops to repeat a process or move through a data collection type.
4. Reusable code - Using either built-in or user-created functions to perform tasks that can be utilized either once or multiple times once they are created.

We also learned how to create variables by using proper naming techniques. Remember our first one? `myVar = "Hello World"`. We have created many variables since then. We also explored various uses of data collection types like lists, sets, tuples, and dictionaries and the behaviors and properties of each.
We have seen and discussed some poorly written code and had hands-on experience with writing good code. And even then we took good code and optimized it. It took three iterations for the Tic-Tac-Toe game but in the end, it was a much better game and provided a much better player experience. Hopefully, at this point in the course, you are starting to see that there is a bit of art and aesthetic to writing code.

And finally, you have likely noticed that our programs are getting longer and longer as we continue to learn new functionality that exists within Python. So it makes sense that the more experienced we become, if the solutions dictate the need, the longer the programs will become.

DID YOU KNOW

Some programs can get to be millions of lines long! The average modern high-end car can contain up to 100 million lines of code. Here is a fun statistics site that references average lines of code per application: www.informationisbeautiful.net/visualizations/million-lines-of-code/

So, why this recap?

Well, a few things:

- For starters, up to this point in the course, we have relied more on functions and logic coding. As the first paragraph stated, we have used those four structures of code (sequential, conditional, repetitive, and reusable), we have defined our own variables and functions, and we have utilized common data collection types. What we are moving to now is a new way to organize our code by bundling objects.
- Second, as our programs get larger, it becomes increasingly important to write code that is easy to understand and structured in a way to optimize it as best as possible. If we are working on a large program, we can never keep the entire program in our mind at the same time. We need to always think of ways to break large programs into multiple smaller pieces ("bundling') so that we have less to look at when solving a problem, fixing a bug, or adding a new feature. You probably noticed that when we optimized our past coding examples and projects, it was performed to reduce repetition, clean up code, and thus help reduce the total amount of lines of code needed. That IS the art and aesthetic to writing code. Keep it simple and clean.
- And finally, here is the best part. All those variables, data collection types, functions, and methods that we have been using are all included in this next model of programming. In fact, it was essential that you knew the basics before we moved to this unit. Because every item of data is an object in Python. Remember when we first called out the term "object"? If not, it was way back in Unit 1 when we were discussing basic built-in functions and methods. We stated that an object is an instance of a class that has properties and methods that are encapsulated or part of it. So up to this point, we have actually been using objects and classes.

That brings us to the term object-oriented programming. You may have heard this term outside of this course (or maybe not). **Object-oriented programming** (or OOP for short) is a programming model that organizes the design of code by bundling objects. Programming languages like Python as well as Java, C++, Ruby, and others are based on the OOP model.

Why use OOP?

The structure and organization of the object-oriented programming model make code more reusable, more scalable, and more efficient. To do this, OOP structures programming into reusable pieces of code (classes) that we can create individual instances of objects from. The key to this challenge is to have a basic understanding of how objects are constructed and how they function, and most importantly, how we make use of the capabilities

of objects that are provided to us by Python. This is an entry-level course in Python so we will touch on the basics of OOP, but know that there is much more capability to it.

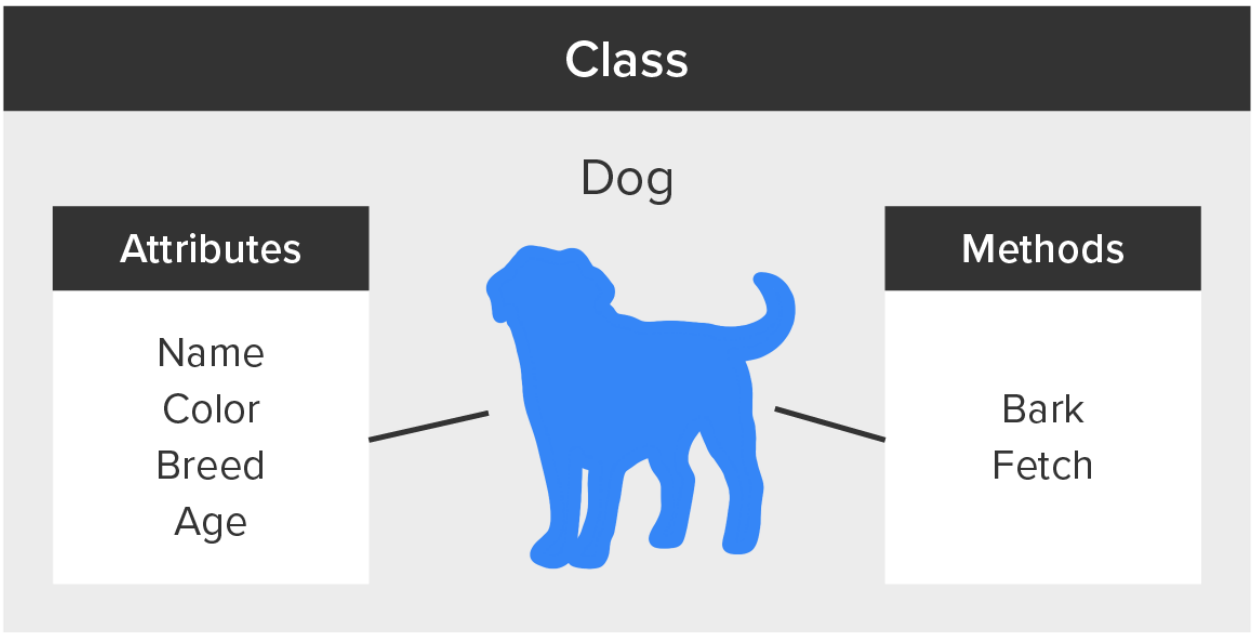
TERM TO KNOW

Object-Oriented Programming

Object-oriented programming (or OOP for short) is a programming model that organizes the design of code by bundling objects.

2. Understanding the Purpose of a Class

What does a class look like? It has a class name, and typically contains attributes and methods.



What is a class?

A **class** is basically a piece of code where we can generate a unique object, much like a template. We can think of classes as a blueprint or a factory in which we can create individual objects using that template. In the example above, `Dog` is a class; however, it doesn't contain any actual data. Classes contain fields for attributes and methods. Our class only contains the attributes and methods (behaviors) that are common to all dogs. The `Dog` class does specify that names, color, age, etc. are important to define a dog and that dogs should have behaviors like barking and fetching, but it does not have any specific data about any particular dog.

Don't worry about the code yet, as we'll get to this in upcoming lessons, but this is an example of a class.

EXAMPLE

```
class Dog:
  def __init__(self,name,breed,age,color):
    self.name = name
    self.breed = breed
    self.age = age
    self.color = color

  def bark(self):
    print("Woof")

  def fetch(self):
    print(self.name," went to fetch.")
```

But how do we use this class?

In order to use a class, we need to create an instance of that class which is called an object. This process is called **class instantiation**. Therefore, objects are an instance of a class that has properties (attributes) and methods that are encapsulated or part of it. The object (instance) is uniquely created from the class and has data attributes that contain values that are unique to it. So, if we created an instance of the `Dog` class, this instance (object) is no longer a blueprint. This instance now contains actual data like a dog's name (Fluffy, for example), who is brown, a beagle, and is 2 years old.

EXAMPLE

```
# an instance of theSnipping Dog class
my_dog = Dog("Fluffy","Beagle",2,"Brown")

print(my_dog.name)
print(my_dog.breed)
print(my_dog.age)
print(my_dog.color)
```

Here is that output screen.

```
Fluffy
Beagle
2
Brown
```

But haven't we been using classes all along? Yes, we have!

Remember that we said that every item in Python is an object. This is true as every item will have methods that can be called from them, including strings or integers.

THINK ABOUT IT

Think back to some of the objects we have used, like strings, and what methods we used on them. That was using classes. A lot of the functionality behind how the string methods are used is completely hidden away from you as the user of the class. We don't have to worry about anything other than the method that we're calling within the class, the parameters, and the return type. This allows us as the developers to just focus on the part of the problem that we have to solve and ignore all of the other parts of the program that are already implemented for us.

Let's see how that works for a variable called '`my_string` using the methods `.upper()` and `.lower()`.

EXAMPLE

```
my_string = "Sophia"
print(my_string.upper())
print(my_string.lower())
```

Now for the output screen.

```
SOPHIA
sophia
```

Without us having to implement those methods to convert the string to lowercase and uppercase, we can simply make use of the existing methods that have been created for this purpose. In doing so, we convert the string all to uppercase characters and separately all to lowercase characters.

So, what is the purpose of a class?

Think of a class like a cookie-cutter and the objects created using the class are the cookies. You don't put frosting on the cookie-cutter; you put frosting on the cookies, and you can put different frosting on each cookie. The frosting in this case represents the attributes or properties within the object. Each individual cookie with the frosting is considered as an object/instance of the class.

In our example, the Dog class (template for creating dogs) will not change but each instance we create from it can. So, we could create other instances like these:

EXAMPLE

```
my_dog1 = Dog("Fluffy","Beagle",2,"Brown")
my_dog2 = Dog("Mochi","Mutt",5,"White")
my_dog3 = Dog("Wolfie","Maltese",10,"Black")
```

In each one of these instances we do not have to redefine the attributes and methods since they were originally defined in the Dog class.

TERM TO KNOW

Class Instantiation

The process of creating an instance of a class which is called an object.

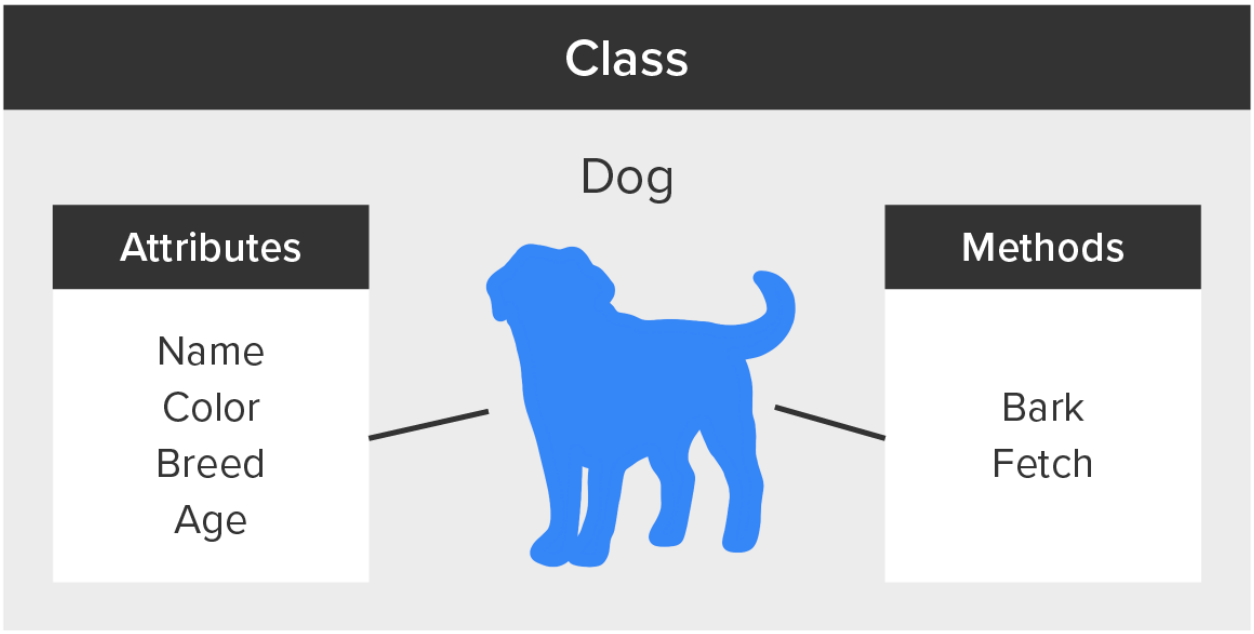
3. Attributes and Methods

Before we discuss the attributes and methods of a class, it is important to bring up the term encapsulation again. We first coined this term in Unit 2 when we were discussing nested functions and a way to bundle functionality into a single unit. Now that we are discussing bundling objects as part of the OOP model, encapsulation is very important to classes as well.

KEY CONCEPT

In object-oriented programming, encapsulation is an important concept. Encapsulation helps describe the concept that we can wrap attributes and methods that work on data in a single class. This allows us to add some restrictions to the direct access of attribute variables and methods. To prevent attributes from being directly changed, a class's attributes can be set to only be changed by its own class's method(s). For example, if we had a numeric grade attribute in a grade book object, we may want to force it to only be set between the numbers 0 and 100. A user should not be able to make a change to allow a negative number or have bonus points as part of the rules of encapsulation.

Back to our class example.



So, we know that a class is a blueprint or template from which objects or instances are created, but we also mentioned that a class contains fields for attributes and methods. Let's dive into those.

Attributes

Attributes can also be known as properties of an object. **Attributes** are variables that are defined in a class template and are the characteristics or properties of an object. In our example, color, breed, and age will be

properties of the class Dog.

Methods

We have used methods previously. Methods are a specialized type of procedure that's directly associated with an object and called to perform a specific task by using the object name, a period, and then the method name. When individual objects are created from the class, these objects can call a method that was defined in the class. Since methods perform actions, a method might change or update an object's data (attribute) or possibly return some information. In our example, some behaviors or actions of the `Dog` class are functions, including barking and/or fetching.

You will get a better understanding of each of the items of a class as we move through this challenge. Understanding the basics of a class is key to utilizing the OOP model.

An Example Class

Next, let's see a class and get to know some of the syntax aspects of it. Here is an example that we will break down.

EXAMPLE

```
class PeopleCounter:
   x = 0

   def anotherOne(self) :
     self.x = self.x + 1
     print("So far",self.x)
```

In the first line of code, we use the `class` keyword.

```
class PeopleCounter:
```

The **class keyword** defines the data and code that will make up each of the objects. When defining a class, it starts with the `class` keyword and is then followed by the name of the class and a colon. In our example above, we are defining a class called `PeopleCounter`.

But that is just the start of declaring a class; we can't just have that line alone. We need to include something about this new class. Notice that after the first line, the rest of the lines are indented. This means they are included in the class. It consists of attributes and methods of that class.

THINK ABOUT IT

Do you notice how this is very similar to defining functions? The definition works in the same way. We start the class by using the keyword class and then the class name followed by a colon.

Indented and defined in the class is a variable we named *x*. We set *x* to 0. This is an attribute of the class; it is defined data for use later. We will get more into this (assigning values to attributes) in a later lesson, including

why this is done and how it is used.

```
class PeopleCounter:
   x = 0

   def anotherOne(self) :
     self.x = self.x + 1
     print("So far",self.x)
```

Next we have declared a method and named it `anotherOne`. Notice the method looks like a function, starting with the `def` keyword, its name, and a colon. Its body consists of its own indented block of code. So, this class has one attribute (`x`) and one method (`.anotherOne()`).

One thing you may have noticed is the word `self` in the method. Methods have a special first parameter that we name by convention `self`.

KEY CONCEPT

`self` is basically a variable that represents the instance of the class (the current object using it). By convention, we use the word `self` only because it is almost an industry standard. You really can use anything in its place. However, most references to methods and search findings will typically have the word `self` utilized. Plus, the name is close to what is actually happening. When we use `self`, we can access the attributes and methods of a class.

Basically this `.anotherOne()` method is taking the attribute *x* and adding 1 to it each time the method is called.

Using the Class

BIG IDEA

So, we defined a class called `PeopleCounter`. What's next? Does something happen if we run the code? No.

Just as the `def` keyword does not cause function code to be executed, the `class` keyword does not create an object. Instead, the `class` keyword defines a template (remember blueprints) indicating what data and code will be contained in each object of the class `PeopleCounter`. There are no executable lines of code so far in this program. Right now we just have the class defined with what attributes and methods we want the instances of this class to use.

Remember the cookie-cutter analogy we used earlier? We described a class as the cookie-cutter and what we create from it are cookies.

Now let's take a look at how we can make a cookie from that cookie-cutter or, in this case, a people instance of the `PeopleCounter` class that we defined above.

EXAMPLE

```
class PeopleCounter:
   x = 0

   def anotherOne(self) :
     self.x = self.x + 1
     print("So far",self.x)

people = PeopleCounter() # creation of an instance of the class PeopleCounter
people.anotherOne()
people.anotherOne()
people.anotherOne()
people.anotherOne()
```

If we were to run this code now, we would see the following output screen.

```
So far 1
So far 2
So far 3
So far 4
```

Let's continue to break down this code—this time, the bottom half.

```
people = PeopleCounter() # creation of an instance of the class PeopleCounter
people.anotherOne()
people.anotherOne()
people.anotherOne()
people.anotherOne()
```

As we continue to look at this program, we see the first executable line of code.

```
people = PeopleCounter()
```

This is where we instruct Python to construct (i.e., create) an object/instance of the class `PeopleCounter`. We chose to name the object created from `PeopleCounter` people. It looks like a function call to the class itself. Python constructs the object with the right attributes (data) and methods (functions) and returns the object which is then assigned to our variable `people`.

When the `PeopleCounter` class is used to construct an object, the variable `people` is used to point to that object. We use the variable `people` to access the attributes and methods for that particular instance of the `PeopleCounter` class.

Each `PeopleCounter` object/instance contains within it a variable `x` and a method named `.anotherOne()`. We call the `.anotherOne()` method on this line:

```
people.anotherOne()
```

Notice that we have now used the standard method call structure using the period (.) before the method name.

When the `.anotherOne()` method is called, the first parameter of that method (remember `self`) points to the particular instance of the `PeopleCounter` class that `.anotherOne()` method is called from (in our case, the `people` instance).

Within the `.anotherOne()` method, we see the line.

```
self.x = self.x + 1
```

This syntax using the dot (.) operator is saying 'the x within self.' The `.` (dot) operator connects the object (instance of a class) to the attributes and methods of that object. So for this first call to the method we have 0 = 0 + 1, so now x = 1. Each time the `.anotherOne()` method is called, the internal `x` value is incremented by 1 and the value is printed out.

Directions: Try adding the code below and see if you get the same results as we did above when running this defined class and instance calls.

```
class PeopleCounter:
   x = 0

   def anotherOne(self) :
     self.x = self.x + 1
     print("So far",self.x)

people = PeopleCounter() # creation of an instance of the class PeopleCounter
people.anotherOne()
people.anotherOne()
people.anotherOne()
people.anotherOne()
```

Attributes

Attributes are data defined in the class template and are the characteristics or properties of an object.

class

The `class` keyword defines the data and code that will make up each of the objects. When defining a class, it starts with the `class` keyword and is then followed by the name of the class and a colon.

.

The (.) (dot) operator connects the object (instance of a class) to the attributes and methods of that object.

SUMMARY

In this lesson, we started off with a **recap** of what we have learned up to this point. We discussed that if we want to have a program that is more scalable, efficient, and especially more reusable, then moving to a model of **object-oriented programming (OOP)** is the way to go. We saw the key **purpose of a class** is to set up this "template" and allow objects (instances) to be created from it. We identified the basics of the class including **attributes** or properties and **methods**. Then, we created a class called `PeopleCounter` that included one attribute and one method.

Best of luck in your learning!

Source: THIS CONTENT AND SUPPLEMENTAL MATERIAL HAS BEEN ADAPTED FROM "PYTHON FOR EVERYBODY" BY DR. CHARLES R. SEVERANCE ACCESS FOR FREE AT www.py4e.com/html3/ LICENSE: CREATIVE COMMONS ATTRIBUTION 3.0 UNPORTED.

TERMS TO KNOW

.
: The (.) (dot) operator connects the object (instance of a class) to the attributes and methods of that object.

Attributes
: Attributes are data defined in the class template and are the characteristics or properties of an object.

Class Instantiation
: The process of creating an instance of a class which is called an object.

Object-Oriented Programming
: Object-oriented programming (or OOP for short) is a programming model that organizes the design of code by bundling objects.

class
: The `class` keyword defines the data and code that will make up each of the objects. When defining a class, it starts with the `class` keyword and is then followed by the name of the class and a colon.