Instance Methods

Introduction

Now that you know what classes and instances are, we can talk about instance methods. Instance methods are almost the same as regular functions in Python. The key difference is that an instance method is defined inside of a class and bound to instance objects of that class. Instance methods can be thought of as an attribute of an instance object. The difference between an instance method and another attribute of an instance is that instance methods are *callable*, meaning they execute a block of code. This may seem a bit confusing, but try to think about instance methods as functions defined in a class that are really just attributes of an instance object from that class.

Objectives

You will be able to:

- Compare instance methods and attributes
- Define and call an instance method
- Define instance attributes
- Explain the `self` variable and its relation to instance objects
- Create an instance of a class

Instance methods as attributes

When you think of methods and functions, you think about what kind of actions they perform. The same goes for instance methods, however, the action being performed is scoped directly to that instance object. Remember, classes are kind of like the blueprints for their instance objects. So, let's take the example of a **Dog** class. What are the things that all dogs do? They can bark, beg to go for a walk, chase squirrels, etc. When you create a new dog instance object, the dog should be able to automatically bark, beg, and chase squirrels.

Let's see how you would create a single dog, `rex`, and get him to bark. First define a `Dog` class:

In [1]:
```
class Dog:
    pass
```

Now we create instantiate this class to create `rex`:

In [2]:
```
rex = Dog()
rex
```

Out[2]:
```
<__main__.Dog at 0x111a83be0>
```

Can `rex` bark?

In [3]:
```
rex.bark()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-d177611622ff> in <module>
----> 1 rex.bark()

AttributeError: 'Dog' object has no attribute 'bark'
```

Okay, here you have an instance of the `Dog` class, but as you can see `rex` cannot bark yet. Let's see if we can fix that. We said that instance methods are basically functions that are ***callable*** attributes, like functions, of an instance object. So, let's write a function that returns the string `"bark!"`, and assign it as an attribute of `rex`.

> **Note:** Dictionary object attributes are accessed using the bracket (`[]`) notation. However, instance object attributes are accessed using the dot (`.`) notation.

In [4]:
```
def make_a_bark():
    return 'ruff ruff!'

rex.bark = make_a_bark
rex.bark
```

Out[4]:
```
<function __main__.make_a_bark()>
```

Here you can see that we successfully added the `bark` attribute to `rex` and assigned the function `make_a_bark` to it. Note that the return value of `rex.bark` is simply a function signature since you have not yet executed the function, and although this looks like an instance method it is not quite.

> **Note:** Although you may hear and see the terms method and function used interchangeably, there are slight differences. You know that a function essentially contains a block of code and it can optionally take in data or objects as explicit parameters, operate on them, and optionally return a value. On the other hand, a method is, simply put, a function that is bound to a class or instances of that class. Instance methods, thus, are functions that are available/bound to instance objects of the class in which they are defined. However, a key difference between the two is that the object is *implicitly* passed to the method on which it is called, meaning the first parameter of the method is the object. Don't worry if this is confusing as you will dive more into this later.

Now you can make `rex` bark by calling the `.bark()` method:

In [5]:
```
rex.bark()
```

Out[5]:
```
'ruff ruff!'
```

This is a great first step. However, since `make_a_bark()` is not actually defined inside our class, you are able to call it without our dog instance object, `rex`, and we mentioned above, that's not really how instance methods work.

In [6]:
```
make_a_bark()
```

Out[6]:
```
'ruff ruff!'
```

Alright, so, how do you turn this into a real instance method? Well, the first thing you need to do is define it inside of our class. So, let's take a look at how you can do that.

Define an instance method

In [7]:
```
class Dog:

    def bark():
        return "I'm an instance method! Oh and... ruff ruff!"
```

In [8]:
```
new_rex = Dog()

new_rex.bark
```

Out[8]:
```
<bound method Dog.bark of <__main__.Dog object at 0x111ab8f98>>
```

Here, we have re-defined our `Dog` class, but this time we actually defined an *instance method* `bark()`. Now, whenever you create a new instance of this new `Dog` class, that instance will have the `bark()` instance method as an attribute.

Notice that the signature that is returned by the unexecuted method says **bound `method`** instead of function, as was the case with our first `rex`'s bark.

However, there is still **one** issue with our instance method. Let's try calling it and see what happens:

In [9]:
```
new_rex = Dog()
new_rex.bark()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-6e1d350549b7> in <module>
      1 new_rex = Dog()
----> 2 new_rex.bark()

TypeError: bark() takes 0 positional arguments but 1 was given
```

Uh oh! `TypeError: bark() takes 0 positional arguments but 1 was given`. This error is telling you that the method, `bark()` was defined to take 0 arguments, but when you executed it, you gave it an argument.

Remember that one of the key differences between functions and methods is that a method is bound to an object and **implicitly** passes the object as an argument. That is what is causing this error. Effectively, what is happening when you try to call the instance method is this:

```
# the instance object, new_rex, is implicitly passed in as the first argument upon execution
new_rex.bark(new_rex)
```

So, how do you fix this error? Well, if instance methods will always require a default argument of the instance object, you will need to define the instance methods with an *explicit* first parameter.

> **Note:** Parameters are the variable names you give to the method or function's future data. They are called parameters when you talk about the definition of a method or function, but when you pass the data, they are referred as arguments.

```
# Since you are defining the function, the variables, parameter1 and parameter2, are called parameters
def function_example(parameter1, parameter2):
    return parameter1 + parameter2

# Here the strings passed in, 'Argument1' and 'Argument2', are arguments since you are passing them
# as data to the function
function_example('Argument1', 'Argument2')
```

Okay, so let's see if you define the method with a parameter, you can get rid of the error. You'll also define another method `who_am_i()` to help further understand what's happening here.

`self` to the rescue!

As with any function or method, you can name the parameters however you want, but the convention in Python is to name this first parameter of all method classes as `self`, which makes sense since it is the object it*self* on which you are calling the method.

In [10]:
```
class Dog:

    def bark(self):
        return 'I am actually going to bark this time. bark!'

newest_rex = Dog()
newest_rex.bark()
```

Out[10]:
```
'I am actually going to bark this time. bark!'
```

Awesome! It works. Again, since instance methods implicitly pass in the object itself as an argument during execution, you need to define each method with at least one parameter, `self`. The concept of `self` is a little confusing. Let's play around with it and see if you can get a better idea. In the following cell we added one more instance method, `who_am_i()` to learn more about `self`:

In [11]:
```
class Dog:

    def bark(self):
        return 'I am actually going to bark this time. bark!'

    def who_am_i(self):
        return self
```

Let's create a new dog (instance), `fido`:

In [12]:
```
fido = Dog()
print("1.", fido.who_am_i()) # Check return value of method
print("2.", fido) # Comparing return of the fido instance object
```

```
1. <__main__.Dog object at 0x111b69ef0>
2. <__main__.Dog object at 0x111b69ef0>
```

As you can see our `who_am_i()` method is returning the same instance object as `fido`, which makes sense because we called this method **on** fido, and if you look at the method all it does is return the first argument (`self`), which is the instance object on which the method was called.

In [13]:
```
# Let's create another rex
another_rex = Dog()

print("3.", fido == fido.who_am_i())
print("4.", another_rex == another_rex.who_am_i())
# again asserting that `self` is equal to the instance object on which who_am_i was called
```

```
3. True
4. True
```

Again, don't worry if `self` still seems a bit confusing. It will become clearer through practice. For now, you can just go forward with the knowledge that **to define an instance method and later call it on an instance object, you will need to include at least one parameter in the method definition, `self`.**

Summary

In this lab, you were introduced to a lot of important concepts in Object-oriented programming. You looked at instance methods and added functions as attributes of objects. Then you looked at the differences between functions and instance methods. You learned that instance methods are bound to objects and they always use the object on which they are called as their first argument. Since instance methods use their object as an argument you looked at how to properly define an instance method by using `self`.