A Deeper Dive into `self`

Introduction

In this lesson, you'll learn a little more about `self` in object-oriented programming (OOP) in Python. You've seen a little bit about `self` when you learned about defining and calling instance methods. So far you've seen that `self` is always explicitly defined as the instance method's **first parameter**. You've also seen that instance methods implicitly use the instance object as the **first argument** when you call the method. By convention, you name this first parameter `self` since it is a reference to the object on which you are operating. Let's take a look at some code that uses `self`.

Objectives

You will be able to:

- Explain the `self` variable and its relation to instance objects

Using `self`

In order to really understand `self` and how it's used, it is best to use an example. Let's use the example of a **Person** class. A class produces instance objects, which in turn are just pieces of code that bundle together attributes like descriptors and behaviors. For example, an instance object of a `Person` class can have descriptors like `height`, `weight`, `age`, etc. and also have behaviors such as `saying_hello`, `eat_breakfast`, `talk_about_weather`, etc.

In [1]:
```python
class Person():

    def say_hello(self):
        return 'Hi, how are you?'

    def eat_breakfast(self):
        self.hungry = False
        return 'Yum that was delish!'

gail = Person()
print('1.', vars(gail))
gail.name = 'Gail'
gail.age = 29
gail.weight = 'None of your business!'
print('2.', gail.say_hello())
print('3.', gail.eat_breakfast())
print('4.', vars(gail))
```

```
1. {}
2. Hi, how are you?
3. Yum that was delish!
4. {'name': 'Gail', 'age': 29, 'weight': 'None of your business!', 'hungry': False}
```

Here you can see that the person instance objects have two behaviors (`say_hello()` and `eat_breakfast`) and you can also add instance variables and assign values to them pretty easily. Additionally, note that you also can add instance variables to `gail` by using `self` inside our instance methods (as in the `eat_breakfast()` method).

Operating on `self`

If you wanted a method that introduces oneself, it would be apt to be similar to the `.say_hello()` method. However, it would also need to include the person's name. To do this, referencing a call to self to retrieve an object attribute is essential.

In [3]:
```
class Person():

    def introduce(self):
        return f'Hi, my name is {self.name}. It is a pleasure to meet you!'

    def say_hello(self):
        return 'Hi, how are you?'

    def eat_breakfast(self):
        self.hungry = False
        return 'Yum that was delish!'

gail = Person()
gail.name = 'Gail'
the_snail = Person()
the_snail.name = 'the Snail'
print('1. ', gail.introduce())
print('2. ', the_snail.introduce())
```

```
1.  Hi, my name is Gail. It is a pleasure to meet you!
2.  Hi, my name is the Snail. It is a pleasure to meet you!
```

Great! See how the method is the same for both instance objects, but `self` is not the same. `self` always refers to the object which is being operated on. So, in the case of `gail`, the method returns the string with the `name` attribute of the instance object `gail`.

Now let's think about some of our other behaviors that might be a bit more involved in order to make them dynamic. For example, everyone's favorite default conversation, the weather. It changes rapidly and seems to always be a choice topic for small talk. How would we create a method to introduce ourselves and make a comment about the weather? Talk about a great way to start a friendship!

Let's see how we would do this with just a regular function:

In [4]:
```
def say_hello_and_weather(instance_obj, weather_pattern):
    return f"Hi, my name is {instance_obj.name}! What wildly {weather_pattern} weather we're having, right?!"

say_hello_and_weather(the_snail, 'overcast')
```

Out[4]:
```
"Hi, my name is the Snail! What wildly overcast weather we're having, right?!"
```

Alright, all is well and good, but let's take a look at how to incorporate this into our class object. Here's an updated version as a class method:

In [5]:
```
class Person():

    def say_hello_and_weather(self, weather_pattern):
        # we are using self instead of instance_obj because we know self represents the instance object
        return f"Hi, my name is {self.name}! What wildly {weather_pattern} weather we're having, right?!"

the_snail = Person()
the_snail.name = 'the Snail'
print('1. ', the_snail.say_hello_and_weather('humid'))
# notice that we are ONLY passing in the weather pattern argument
# instance methods **implicitly** pass in the instance object as the **first** argument
```

```
1.  Hi, my name is the Snail! What wildly humid weather we're having, right?!
```

Again, note that the only arguments you pass in are those that come after `self` when you define an instance method's parameters.

Now that you've seen how to leverage `self` and even use instance methods with more than just `self` as an argument, let's look at how you can use `self` to operate on and modify an instance object.

Let's say it is `gail`'s birthday. Gail is 29 and she is turning 30. To ensure the instance object reflects that you can define an instance method that updates `gail`'s age:

In [6]:
```
class Person():

    def happy_birthday(self):
        self.age += 1
        return f"Happy Birthday to {self.name} (aka ME)! Can't believe I'm {self.age}?!"

the_snail = Person()
the_snail.name = 'the Snail'
the_snail.age = 29
print('1. ', the_snail.age)
print('2. ', the_snail.happy_birthday())
print('3. ', the_snail.age)
```

```
1.  29
2.  Happy Birthday to the Snail (aka ME)! Can't believe I'm 30?!
3.  30
```

While this method could be improved, the important note is `self` can be used to not only *read* attributes from the instance object, but can also change the attributes of the instance object. `self` is simply the means by which to access underlying attributes stored within the object whether you want to retrieve said information or update it.

Let's take this a step further and look at how you can call other methods using `self`.

Calling Instance Methods on `self`

Another very important behavior people have is eating. It is something that we all do and it helps prevent us from getting **hangry**, or angry because we're hungry.

In [7]:
```python
class Person():

    def eat_sandwhich(self):
        if (self.hungry):
            self.relieve_hunger()
            return "Wow, that really hit the spot! I am so full, but more importantly, I'm not hangry anymore!"
        else:
            return "Oh, I don't think I can eat another bite. Thank you, though!"

    def relieve_hunger(self):
        print("Hunger is being relieved")
        self.hungry = False

the_snail = Person()
the_snail.name = 'the Snail'
the_snail.hungry = True
print('1. ', the_snail.hungry)
print('2. ', the_snail.eat_sandwhich())
print('3. ', the_snail.hungry)
print('4. ', the_snail.eat_sandwhich())
```

```
1.  True
Hunger is being relieved
2.  Wow, that really hit the spot! I am so full, but more importantly, I'm not hangry anymore!
3.  False
4.  Oh, I don't think I can eat another bite. Thank you, though!
```

Awesome! Be sure to observe that you can also use `self` to call other instance methods (as with the `self.relieve_hunger()` call above).

Summary

In this lesson, you examined how to use `self` in OOP. You first reviewed using `self` to define instance methods appropriately. Next, you saw how to leverage `self` in order to make instance methods a bit more re-usable and dynamic. That is, you saw how you can retrieve object attributes using `self`. You also looked at using multiple arguments in a method call and using `self` to change the attributes on an instance object. Finally, you saw how to use `self` to call other instance methods.