Lecture 24

Object-Oriented Programming

6 Floréal, Year CCXXX

Song of the day: ***Oboe Sonata in D, Op. 166 — Molto Allergo*** *by Camille Saint-Saëns (1921), performed by The Nash Ensemble (2005).*

Part 1: *Programming Paradigms*

So far, we've been thinking of this class in pretty heavily in terms of functions. A function is simply a grouping of functionality so that we, the programmers, might not have to repeat ourselves over and over while performing common processes.

Let's think a bit closer to reality for a bit, though. Let's say, for instance, we're programming an **RPG**, where we have a party of four characters with the following stats:

Name	Health	Attack	Defense
`"Ness"`	`100`	`50`	`50`
`"Paula"`	`120`	`70`	`40`
`"Jeff"`	`80`	`45`	`70`
`"Po"`	`110`	`75`	`35`

Table 1*: Party member stats.*

How might we store this information with the structures that we've worked with so far? Maybe a dictionary?

```
party = {
    "Ness": {
        "Health": 100,
        "Attack": 50,
        "Defense": 50
    },
    "Paula": {
        "Health": 120,
        "Attack": 70,
        "Defense": 40
    },
    "Jeff": {
        "Health": 80,
        "Attack": 45,
        "Defense": 70
    },
    "Po": {
        "Health": 110,
        "Attack": 75,
        "Defense": 35
    }
}
```

That wouldn't be too bad. That way, if we're attacking an enemy, all we have to do is select the name of the character we want to use:

```
enemies["Starman"]["Health"] -= party["Paula"]["Attack"]
```

This doesn't work too bad. What if we wanted to add an additional attribute to each of our party members—something like `"luck"` ? Well, we'd have to iterate through our dictionary and add it to each member individually:

```
def add_character_attribute(party, attribute_name, attribute_value):
    for member in party:
        party[member][attribute_name] = attribute_value

add_character_attribute(party, "luck", 20)
```

This is inefficient for a number of reasons, but the chief reason is that each member will have their luck initialized to the same value of 20. We would *still* need to go through each of our members and manually adjust their values to better fit their attributes:

```
party["Ness"]["Luck"] = 40
party["Paula"]["Luck"] = 50
party["Jeff"]["Luck"] = 20
party["Po"]["Luck"] = 30
```

That looks like...a lot of repetition. Repetition that we can't really avoid, in this case. Whichever way we decompose this, we'll still have to manually build each character up. And this is only four characters we're talking about. Imagine that you had 1,000 enemies in your game, ready to be released, when you decided to give them each an additional attribute? That's 1,000 lines of code to do the exact same thing in a slightly different way.

Not only that, but our `add_character_attribute()` function can be used for literally any dictionary:

```
add_character_attribute(levels, "luck", 20)
```

It doesn't make any sense for your game's levels to have a luck attribute—but if they're a dictionary, then this function will work.

Like I said: all of this *has* workarounds. But at a certain point it becomes evident that function-oriented programming doesn't cut it for every task that we might come across. So what should we do?

Part 2: *Object-Oriented Programming (OOP)*

Turns out we already know the solution for this: objects. We've seen objects before *everywhere* in this class so far. Strings, for instance have methods that only belong to them:

```
names = ["Ayumu", "Setsuna", "Shizuku"]

for index in range(len(names)):
    names[index] = names[index].upper()  # all strings can take advantage of this

print(names)
```

Output:

```
['AYUMU', 'SETSUNA', 'SHIZUKU']
```

Strings also come with their own methods and attributes associated with them:

```
names = ["Ayumu", "Setsuna", "Shizuku"]

for name in names:
    print(name.__class__)  # all strings can take advantage of this attribute
```

Output:

```
<class 'str'>
<class 'str'>
<class 'str'>
```

It would be great if we could do something similar with our party members—that is, have specific attributes and methods that only belong party members and party members alone. Enemies would have their own set, levels would have their own set, etc. This type of code organization, based on objects, is what we call **object-oriented programming**.

In other words, instead of only using Python's built-in types– `int` , `str` , `list` , `dict` , etc. —we can create our own types.

The way we create our own types—or ***classes***, as we call them in OOP—is the following:

```
class ClassName:
    # Class definition here
```

This is basically the bare minimum that we would need. Let's try creating a class for our party members and call it " `Character` ":

```
# Defining our Character class
class Character:
    pass

# Creating/instantiating a Character object
protagonist = Character()

print(protagonist)
```

Output:

```
<__main__.Character object at 0x7ff548194e20>
```

The way I would read this output is:

> In the `__main__` function, there is a `Character` object at memory location `0x7ff548194e20` .

Now, before we go forward, let's talk about some very important terminology that we will be using from this class on:

> ***Class***: The *definition* set of variables and functions that each object of that class will have. Custom class names should always start with a capital letter and then **camelcase**. *Classes do **not** run code*.
>
> ***Object***: An *instance* of a class.

For example:

```
number = 1
string = "Taeyeon"
lst = [number, string]

print("- {} is an object instance of the {}.".format(number, number.__class__))
print("- {} is an object instance of the {}.".format(string, string.__class__))
print("- {} is an object instance of the {}.".format(lst, lst.__class__))
```

Output:

```
- 1 is an object instance of the <class 'int'>.
- Taeyeon is an object instance of the <class 'str'>.
- [1, 'Taeyeon'] is an object instance of the <class 'list'>.
```

The same would work with our custom-made classes:

```
# Defining our Character class
class Character:
    pass

# Creating/instantiating a Character object
protagonist = Character()
print("{} is an object instance of the {}.".format(protagonist, protagonist.__cla
```

Output:

```
<__main__.Character object at 0x7ff530379910> is an object instance of the <class
```

Since our class is still very simple, the output doesn't look quite as nice (we'll fix that later), but what this can simplify to is:

> The object inside the variable `protagonist` is an object instance of the `<class 'Character'>`.

In other words, when you want to create an integer, you don't write `number = int()` —you just give it its starting value and Python looks to the `int` class definition to do the rest.

Let's actually put some contents into our `Character` class. We know that each character has a name, a health stat, an attack stat, and a defense stat. Let's start with only the name. Here is the syntax for this:

```
class Character:
    def __init__(self, name):
        self.name = name
```

As you can see, we defined a function inside the class definition. **Any function defined inside a class definition is called a method** (hence `list` methods, `str` methods, etc.). The `__init__()` method is by far the most important method, and is the only one that is not optional.

Why? Because `init` stands for "initializer" or "initialization". In other words, this method is the one that takes care of giving your object its initial values. Let's see it in practice:

```
class Character:
    def __init__(self, name):
        self.name = name

protagonist = Character("Ness")

print(protagonist.name)
```

Output:

```
Ness
```

Awesome. So, from now on, you will have to give `Character()` a value for name. If you don't, you'll get an error:

```
class Character:
    def __init__(self, name):
        self.name = name
```

```
protagonist = Character("Ness")
boss = Character()
```

Output:

```
Traceback (most recent call last):
  File "<input>", line 6, in <module>
TypeError: __init__() missing 1 required positional argument: 'name'
```

Literally: you forgot to enter a value for the `name` attribute. Let's add in the rest of our attributes to the definition:

```
class Character:
    def __init__(self, name, health, attack, defense):
        self.name = name
        self.health = health
        self.attack = attack
        self.defense = defense

protagonist = Character("Ness", 100, 50, 50)
print(protagonist.name)
print(protagonist.health)
print(protagonist.attack)
print(protagonist.defense)
```

Output:

```
Ness
100
50
50
```

So, what is `self` ?

Sigh. I hate explaining this part, because it's going to be confusing regardless of how I put it. The technical definition for the `self` parameter is:

> `self` : A reference to the newly created object instance of this class.

In other words, `self` represents the object **inside itself**. The best analogy I can come up with is thinking of `self` as your brain, and of your whole body as the whole object. Of course, your brain isn't your whole body, but it is the part of your brain that stores all the information and functionality needed for the rest of your body to function.

So, in this `__init__()` method:

```
class Character:
    def __init__(self, name, health, attack, defense):
        self.name = name
        self.health = health
        self.attack = attack
        self.defense = defense
```

We're basically saying:

> In this instance of the `Character` class, the character's attribute `name` will have the value of the parameter `name`, the attribute `health` will have the value of the parameter `health`, the attribute `attack` will have the value of the parameter `attack`, and the attribute `defense` will have the value of the parameter `defense`.
>
> Beyond this point, I will save these values inside the `self`. So if you want to use them, you'll have to have `self` available.

Let me show you an example of this. Let's say we wanted each `Character` object to have a method that prints its current health:

```
class Character:
    def __init__(self, name, health, attack, defense):
        self.name = name
        self.health = health
        self.attack = attack
        self.defense = defense

    def get_health():
        print("{} has {}pp remaining.".format(name, health))


protagonist = Character("Ness", 100, 50, 50)
protagonist.get_health()
```

Output:

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
```

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

This is confusing. It is telling us that `get_health()` takes 0 arguments (which is exactly how we defined it), but apparently 1 was given at some point. This hidden parameter is `self`.

```
class Character:
    def __init__(self, name, health, attack, defense):
        self.name = name
        self.health = health
        self.attack = attack
        self.defense = defense

    def get_health(self):
        print("{} has {}pp remaining.".format(name, health))
```

Remember that if you consider `self` as your brain, every other part of your body needs to be connected to it in some way. Attempting to create that `get_health()` method without including the self would be the equivalent to saying:

> This human will have an arm, but the brain won't be able to control it.

This makes absolutely no sense, so Python doesn't even allow it. We're not done with the errors though. If we added the `self` and attempted to run this again, we'd get this error:

```
class Character:
    def __init__(self, name, health, attack, defense):
        self.name = name
        self.health = health
        self.attack = attack
        self.defense = defense

    def get_health(self):
        print("{} has {}pp remaining.".format(name, health))


protagonist = Character("Ness", 100, 50, 50)
protagonist.get_health()
```

Output:

```
Traceback (most recent call last):
  File "<input>", line 13, in <module>
  File "<input>", line 9, in get_health
NameError: name 'health' is not defined
```

So what's this one telling us? `health` is not defined. You might think that this doesn't make sense (and that's why OOP in Python is so confusing at first), but read the last line that I wrote earlier:

> Beyond this point, I will save these values inside the `self`. So if you want to use them, you'll have to have `self` available.

So `health` *is* defined, but *inside* the `self`. To access it, you just need to write `self.health`:

```
class Character:
    def __init__(self, name, health, attack, defense):
        self.name = name
        self.health = health
        self.attack = attack
        self.defense = defense

    def get_health(self):
        print("{} has {}pp remaining.".format(self.name, self.health))


protagonist = Character("Ness", 100, 50, 50)
protagonist.get_health()
```

Output:

```
Ness has 100pp remaining.
```

Perfect. Let's define a couple of other methods that we talked about: attacking.

```
class Character:
    def __init__(self, name, health, attack, defense):
        self.name = name
        self.health = health
        self.attack = attack
        self.defense = defense

    def get_health(self):
        print("{} has {}pp remaining.".format(self.name, self.health))
    def attack_enemy(self):
        return self.attack


protagonist = Character("Link", 100, 50, 50)
final_boss = Character("Ganon", 200, 50, 50)

final_boss.health -= protagonist.attack_enemy()
final_boss.get_health()
```

Output:

```
Ganon has 150pp remaining.
```

These are the very basics of OOP. Next week we'll get into more functionalities we can take advantage of.