Lecture 25

Object-Oriented Programming

8 Floréal, CCXXX

Song of the day: ***Hikaru Toki*** *by Hitsuji Bungaku (2022).*

Part 1: *OOP Review*

The `__init__()` ***Method***

In the file **music.py**, create a **class** called `Song` that will be created by the user with the following attributes:

Attribute	Type
`song_title`	`str`
`artist`	`str`
`album`	`str`
`genre`	`str`
`length_in_seconds`	`int`

Table 1: Attributes passed in by the user to create a `Song` object.

See the following sample behavior below showing the creation of a `Song` object called `a_random_song` :

```
a_random_song = Song("The Girls Are Alright!", "saya", "The Girls Are Alright! – EP", "Indie", 271)
```

In addition to these 5 variables, inside the `Song` class's `__init__` , define a 6th attribute called `play_count` . The user needn't pass this variable in, as all songs begin with a play count of 0. Once your `__init__` is properly implemented, your class should behave as follows:

```
a_random_song = Song("The Girls Are Alright!", "saya", "The Girls Are Alright! – EP", "indie", 271)
print(a_random_song.song_title)
print(a_random_song.artist)
print(a_random_song.album)
print(a_random_song.genre)
print(a_random_song.length_in_seconds)
print(a_random_song.play_count)
```

Output:

```
The Girls Are Alright!
saya
The Girls Are Alright! – EP
Indie
271
0
```

The `play()` Method

Then, then define a method called `play()`. Quite simply, when this method is called, object's `play_count` will be increased by 1. It does not accept any parameters:

```
a_random_song = Song("The Girls Are Alright!", "saya", "The Girls Are Alright! – EP", "indie", 271)
num_of_plays = 10
for time in range(num_of_plays):
    a_random_song.play()

print(a_random_song.play_count)
```

Output:

```
10
```

Part 2: *Anatomy of a Method Invocation*

Last time, we left our `Character` class's definition looking like this:

```
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
```

Code Block 1: Our `Character` class, **currently**.

So, essentially:

- Four (4) attributes: `name`, `health`, `attack`, `defense`.
- Two (2) methods: `get_health()`, `attack_enemy()`.

When we instantiate (i.e. create an instance/object of this class/type, we would do something like this:

```
protagonist = Character("Kasane Randall", 100, 142, 99)
```

Part-by-part, it is:

- `protagonist` : The namespace reference (variable) to this `Character` object.
- `Character` : Reference to the `Character` class.
- `(...)` : The instantiation operator i.e. the operator that creates an instance of this class.

In order to instantiate our object with some initial values (i.e. `"Kasane Randall", 100, 142, 99`), our class **definition** would need an `__init__()` method:

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

That is, the string `"Kasane Randall"` is passed into the initializer and assigned to the `Character` object's `name` attribute (`self.name` **inside** the class definition), `100` will be assigned to the `health` attribute, etc. If you do not define an `__init__()` method, the class will still be created, but without any instance variables:

```
class Character:
    pass


empty_character = Character()  # this does not fail
print(empty_character)
```

Output:

```
<__main__.Character object at 0x7f9b50093c70>
```

That is, an object of the `Character` class exists at memory location `0x7f9b50093c70` .

Now, as a quick reminder of what a method is, it's just a **function that is bound to an object of a specific class**. Common examples that we've used in class are string methods such as `split()` and `strip()` . This means that the `str` class definition probably looks like this:

```
class str:
    ...
    ...
    def split(self, separator=' '):
        # method definition here
    ...
    ...
    def strip(self, character):
        # method definition here
    ...
    ...
```

Just like `split()` and `strip()` , we could call out `Character` instance methods by using the dot `.` operator:

```
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("Paul McCartney", 100, 50, 50)
protagonist.get_health()
```

Output:

```
Paul McCartney has 100pp remaining.
```

When we make a call to `get_health()`, Python looks at the method definition inside the `Character` class's definition, and `protagonist.get_health()` becomes `self.get_health()`. Now, `get_health()` makes use of this object's `name` and `health` attributes, which we can access within the class definition using `self.name` and `self.health`.

Part 3: *Special Methods*

So, now we've reached a point where the following behavior bothers me:

```
title = "Purple"
artist = "Lil Wayne"
album = "Jeff"
genre = "Hip-hop"
length = 123

new_song = Song(title, artist, album, genre, length)
print(new_song)
```

Output:

```
<__main__.Song object at 0x7f9ea81282e0>
```

I don't particularly care about the name of the file in which this class is being defined in, nor do I care about its exact location in my computer's memory. I care a lot more about the `Song` object's title and artist. This information is a lot more pertinent to what this object is supposed to represent—an irl song.

So how can we change the behavior of our custom-made objects so that they are formatted nicely when we print them?

The answer to this question lies in **special methods**, of which there are many. Today, we will look at `__str__()`.

The `__str__()` Method

In order get us a nice *string representation* of an object, we need to define its behavior when being **casted into a string**. That's what the `__str__()` method defines:

```
class Song:
    ...
    ...
    def __str__(self):
        """
        Returns informal representation of Song object.
        :return: A string
        """
        string_representation = ________________________  # however we choose to "stringify" our object

        return string_representation
```

The question we must now ask ourselves is:

> When I print a `Song` object, what information do I want showing up?

Unless you have specific instructions from your boss (or the final exam's prompt), this is largely up to you. How about we include its title, artist, and album?

```
def __str__(self):
    string_representation = "{}, by {} ({})".format(self.song_title, self.artist, self.album)
    return string_representation
```

Let's see it in action now:

```
new_song = Song(song_title="Maxwell's Silver Hammer", artist="The Beatles", album="Abbey Road", genre="Classi
                length_in_seconds=200)
print(new_song)
```

Output:

```
Maxwell's Silver Hammer, by The Beatles (Abbey Road)
```

Much nicer.

Part 4: *Program Structure*

Classes must be defined before use, just like functions do:

```
protagonist = Character()

class Character:
    pass
```

Output:

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'Character' is not defined
```

Also, a common convention is to define one class per file. For example, let's say we had a `Weapon` class in **weapon.py**:

```
from random import random


class Weapon:
    def __init__(self, name, power):
        self.name = name
        self.power = power
        self.brittleness = round(random(), 2)

    def get_power_boost(self):
        return round(self.power * self.brittleness, 2)
```

Let's modify our `Character` class to get a `Weapon` object attribute as well:

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


weapon = Weapon("Master Sword", 42)
protag = Character("Link", weapon, 100, 50, 50)
```

Output:

```
Traceback (most recent call last):
  File "<input>", line 16, in <module>
NameError: name 'Weapon' is not defined
```

This makes sense; if we don't import the `Weapon` class definition into **character.py**, it will have absolutely no idea of what it is. So let's do that:

```
from weapon import Weapon


class Character:
    def __init__(self, name, weapon, health, attack, defense):
        self.name = name
        self.weapon = weapon
        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 + self.weapon.get_power_boost()


def main():
    weapon = Weapon("Master Sword", 42)
    protag = Character("Link", weapon, 100, 50, 50)

    print("{} attacks enemy with {} power!".format(protag.name, protag.attack_enemy()))


if __name__ == '__main__':
    main()
```

Possible output:

```
Link attacks enemy with 92.42 power!
```