Joseph Watts
March 12, 2025
IT FDN 110 A Wi 25: Foundations Of Programming Python
Assignment07
GitHub Repository: https://github.com/JBWpro/IntroToProg-Python-Mod7

# Basics of Python: More on Classes

## Introduction

Last week we started diving into classes, using them to format our functions. Now we dive deeper into classes and how we can use them to organize objects.

## Data Classes

Previously we were using classes as processing classes. The focus of the processing class was to perform actions, while the focus of the new data class is to organize data. Data classes are typically made up of attributes, constructors, properties and methods.

## Attributes

An attribute is a piece of data associated with an object. They can be thought of as a characteristic, storing information about the object. Each attribute of an object can have different data types. Attributes can be marked as private, meaning they cannot be changed by code outside of the class. You can mark an attribute as private by adding 2 underscores before the attribute's name. Python does not actually enforce this privacy, but it allows you or anyone else working in the class to recognize that the attribute is private.

## Constructors

A constructor is a method or function that gets called when a class object is created. When a class object gets created, the constructor sets the object's attributes to a set initial value. If you do not create a constructor, python will create an invisible default constructor. In python the constructor is named __init__. Constructors do not have a return type, instead setting the values globally or by using properties. When using constructors, it is common to see the “self” keyword. This is so that when you are using you class objects and functions, the self keyword allows you to reference the data or functions found within the object, not the data or functions within the original class. Here is an example of an constructor using .self:

```
def __init__(self, first_name: str = "", last_name: str = ""):
   self.first_name = first_name
   self.last_name = last_name
```

## Properties

Properties are functions designed to manage attribute data. Without properties, you must use both the .get and .set function each time you want to modify your attributes. Instead, we will now be using two properties, one for getting and one for setting, to accomplish the same thing. These are commonly known as Getters and Setters.

## Getter

An getter allows you to access data and gives you the option of applying formatting. You use the @property decorator to indicate your getter as shown bellow:

```python
@property
def first_name(self):
   return self.__first_name.title()
```

Not only can you see the use of the decorator, but you can also see the use of the double underscores to show the attribute is private.

## Setters

A setter property function allows you to add validation and error handling. A setter is like any other function, with the addition of a decorator. The decorator must include @property_name.setter like so:

```python
@first_name.setter
def first_name(self, value: str):
   if value.isalpha() or value == "":
       self.__first_name = value
   else:
       raise ValueError("The last name should not contain numbers.")
```

Make sure that both your getter and setter functions have the same name, allowing them to form a getter and setter pair. As you can see, a setter is like any other function and will allow you to add error handling directly into the object class.

## Inherited Code

We have made a data class for a person, but what if you also wanted a data class for a student? You could start from scratch, but why do that when you already made a person class that likely shares attributes, such as first and last name. That is where inheritance comes in. A new class, or child class, can inherit data properties and methods from an already existing class. For example, here is our “parent class”:

```python
class Person:
```

And then we can have our student inherent our Person class properties like so:

```python
class Student(Person):
```

When creating a constructor in a child class, you can use super().__init__() to call the parent class’s constructor. This ensures that the attributes from the parent class are properly initialized without needing to redefine them in the child class. For example:

```python
def __init__(self, first_name: str = "", last_name: str = "", course_name: str
= ""):
   super().__init__(first_name=first_name, last_name=last_name)
```

```
    self.course_name = course_name
```
Here you can see that the student object is using super.__init__  to inherit the parent classes constructor for first and last name, while also adding its own attributes lille course name.

## Running the Script

Running the script through either IDLE or the commands will first import the json module. Then it will read all the class and function definitions we have. Then it will read the file "Ennrolments.json" that sits within your directory. Using the json module, the data will be appended from the JSON file to the students_data using the new Student object class. After the data is read a while loop starts. It will print a menu of options, and prompt the user to enter an option from the menu. If they select the first option, they will be asked to enter their first and last name as well as the course name. After that the while loop starts again. If they then pick the second option the script will print the names of all the registered students and their class, including any names you may have just entered. If the third option is selected, the data is written to a file and the data being saved is printed into the terminal. Again we are using the json module to write our dictionary to our json file. Selecting option four exits the program. You may choose any of these options as many times as you like. Entering a number not between 1 and 4 will remind the user that they must choose one of those options.

## Summary

This week's assignment expands on classes by introducing data classes, which organize data using attributes, constructors, properties, and methods. It explains how getters and setters manage private attributes and how inheritance allows a child class to reuse a parent class's attributes and methods using super().__init__(). By utilizing object-oriented programming, we can improve data organization, reusability, and maintainability.