Assignment 07: Classes and Objects

Patrick Renie
Foundations of Programming: Python

08/26/25

1 Introduction

In this module, we learned about classes—one of the fundamental building blocks of object-oriented programming. Classes are complicated stuff, so this module touched on topics like object instances, constructors, properties (getters and setters), magic methods, and more.

2 Classes

Most programs consist of three basic building blocks: **statements**, **functions**, and **classes**. We already know a lot about statements and functions. In this module, we learned more about classes.

2.1 Class-Specific Terminology

Depending on the context, familiar concepts are called different things. This is particularly true in the context of classes. For example, a function within a class is typically called a method. Put another way, a method is a function that's associated with a specific object. For example, `print()` is a function; `capitalize()` is a method of string objects. Similarly, **variable**s and **constant**s in a class are called **fields**, **attributes**, or **properties** depending on how you define and manage them.

2.2 Using Classes to Create Objects

Python is an *object-oriented programming language*. Almost everything in Python is an **object**, including numbers, strings, dictionaries, lists, functions, and even classes. In addition to their role as an organization tool, classes can be used to create new **object instances**.

2.3 Instances

What's an instance? An instance is a particular instance (for lack of a better word) of a class object. An instance might also be called an object instance or class instance; the meaning is the same regardless.

By way of example, say we have the class `Vegetable`. Every instance of the `Vegetable` class has these attributes: `name`, `color`, and `calories`. We can define the `Vegetable` class and create an instance of the class called `carrot` like so:

```
>>> class Vegetable:
...     name: str = ''
...     color: str = ''
...     calories: int = 0
...
>>> carrot = Vegetable()
>>> carrot.name = 'carrot'
>>> carrot.color = 'orange'
>>> carrot.calories = 40
>>> print(carrot, carrot.name, carrot.color, carrot.calories)
<__main__.Vegetable object at 0x000001768A818AD0> carrot orange 40
```

By assigning `Vegetable()` to the variable `carrot`, we create an *instance* of the `Vegetable` class. When we print `carrot`, Python outputs the object's data, including its *memory address*. Each object in Python has its own memory address. We can demonstrate this by reassigning the `carrot` variable and creating a new instance of the `Vegetable` class.

```
>>> class Vegetable:
...     name: str = ''
...     color: str = ''
...     calories: int = 0
...
>>> carrot = Vegetable()
>>> carrot.name = 'carrot'
>>> carrot.color = 'orange'
>>> carrot.calories = 40
>>> print(carrot, carrot.name, carrot.color, carrot.calories)
<__main__.Vegetable object at 0x000001768A818AD0> carrot orange 40
>>> carrot = Vegetable()
>>> carrot.name = 'carrot'
>>> carrot.color = 'orange'
>>> carrot.calories = 40
>>> print(carrot, carrot.name, carrot.color, carrot.calories)
<__main__.Vegetable object at 0x000001768A77E990> carrot orange 40
```

Notice that the memory address of the `carrot` object has changed. That's because we created a new instance of the object, and each instance has its own memory address. This is true even when we use the same variable name and assign the object the same attributes, as in this example. This concept is known as **data encapsulation** and is a core aspect of object-oriented programming.

3 Data Classes

In the previous module we looked at **processing classes**. These are classes designed to process data, such as the `FileProcessor` and `IO` classes we created in Module 06. Usually, processing classes (as well as presentation classes) contain only methods.

Data classes, on the other hand, might contain methods as well as **attributes**, **constructors**, and **properties**.

3.1 Attributes

An attribute is used to store data about the object it belongs to. It is essentially an object-specific variable. For example, `Vegetable.color` is an attribute (that is, `color` is an attribute of the `Vegetable` class).

3.2 Object State

In programming, an object's **state** is the current status of the object's attributes and properties at a particular point in time. An object's state might change many times throughout the course of running a program.

3.3 Constructors

A **constructor** is the programming term for a special method invoked when an object instance is created. Constructors make it much easier to assign attributes to objects because they initialize such attributes at the time of an object's creation.

To create a constructor, all you need to do is define a class's `__init__` method like so:

```
>>> class Fruit:
>>>     """
>>>     This class represents a fruit.
>>>     """
>>>     def __init__(self, name, color, calories):
>>>         self.name: str = name
>>>         self.color: str = color
>>>         self.calories: int = calories
```

In this example, when an object instance of the `Fruit` class is created, it now requires three attributes: `name`, `color`, and `calories`. So a class object declaration would look like this:

```
>>> apple = Fruit('apple', 'red', 100)
```

If we try to create a `Fruit` object without the required parameters, we receive an error:

```
>>> orange = Fruit()
Traceback (most recent call last):
  File "C:\Users\Patrick\fruit_classes.py", line 1, in <module>
    orange = Fruit()
TypeError: Fruit.__init__() missing 3 required positional arguments: 'name',
     'color', and 'calories'
```

We can avoid this error by assigning default attributes to each constructor parameter. We do this the same we assign default arguments to a function. Here's how that would look for our example `Fruit` class.

```
>>> class Fruit:
>>>     """
>>>     This class represents a fruit.
>>>     """
>>>     def __init__(self, name = "", color = "", calories = 0):
>>>         self.name: str = name
>>>         self.color: str = color
>>>         self.calories: int = calories
```

Now, if we define a `Fruit` object without any attributes, the object will have the default attributes we defined in the class declaration. We can assign values to these attributes later using *dot notation*, like so:

```
>>> orange = Fruit()
>>> print(orange, orange.__getstate__())
<__main__.Fruit object at 0x0000026B464C4CD0> {'name': '', 'color': '',
     'calories': 0}
>>> orange.name = 'orange'
>>> orange.color = 'orange'
>>> orange.calories = 50
>>> print(orange, orange.__getstate__())
<__main__.Fruit object at 0x0000026B464C4CD0> {'name': 'orange', 'color':
     'orange', 'calories': 50}
```

Like the `__init__` constructor method, `__getstate__` is a special class method that returns the current state of the class object.

3.4 The self Keyword

When used as a constructor, a class declaration always includes the `self` keyword as the first parameter. (This is kind of like a "secret" parameter, since you can't pass an argument to it.) The `self` keyword refers to the class's object instance, not the class. So, if you have a `Dog` class and create an object instance named `sparky`, the `self` keyword would refer to `sparky`. If you created more dogs, each would get its own `self` keyword too (though you only need to declare `self` once, in the class definition).

You don't need to (and shouldn't!) include the `self` keyword for static methods marked with the `@staticmethod` decorator, since those methods aren't constructors and thus aren't associated with object instances.

3.5 Private Attributes

When naming class methods and instance variables, you can include a leading underscore (e.g., `ClassName._method()`) to mark it as *private*. Private objects are also called "protected" or "non-public" objects.) See PEP 8 and the Python documentation for more details about this convention.

When you mark a variable (or whatever) as private, you're telling yourself and other developers, "Hey, don't call this variable from outside its class!" Some languages enforce this logic by throwing an error if the developer tries to call a private variable from outside its class. Python, idiosyncratically, does not. IDEs such as PyCharm do show the user a warning if they call a private variable outside its class, but the user can choose to ignore the warning if they wish.

3.5.1 Single vs. Double Underscore Attributes

Prepending a name with a single underscore marks it as private. Prepending with *two* underscores activates Python's **name mangling** functionality. A "mangled" name is like a private name, but its "actual" identity is further obfuscated by Python. Say we have class `X` and attribute `y`; to call `y`, you'd just write `X.y`. But if in class `X` we define attribute `__z`, we can call it only by passing its mangled name: `X._X__z`. Python

automatically mangles the attribute's name so that it is prepended with the name of its class preceded with an underscore. This is true whether the mangled name belongs to a class attribute or an instance attribute.

Did You Know?

The double-underscore is sometimes called a "dunderscore." Programmers love expediency!

Here is an example of code, with comments, that illustrates how name mangling works in practice:

```
class Cats:
    def __init__(self, fave_food):
        self.fave_food = fave_food
        self._fave_food = fave_food
        self.__fave_food = fave_food

bugsy = Cats(fave_food='sardines')

print(bugsy.fave_food)
print(bugsy._fave_food)
# These both print "sardines"

try:
    print(bugsy.__fave_food)
except Exception as e:
    print(e)
# returns an error; no such attribute exists

print(bugsy._Cats__fave_food)
# prints "sardines"; the dunderscored attribute is read via its mangled name

bugsy.__fave_food = 'counterfeit sardines'
# Creating a dunderscored attribute outside a class definition
# results in weird behavior. Python creates a new instance attribute
# called, confusingly, __fave_food. So, now, two attributes with similar names
# are accessible: bugsy._Cats__fave_food and bugsy.__fave_food.

print(f"Bugsy's favorite food is {bugsy.__fave_food}")
# prints "Bugsy's favorite food is counterfeit sardines" because the __fave_food
# attribute was just created and assigned a value

print(f"Bugsy's favorite food is {bugsy._Cats__fave_food}")
# still prints "Bugsy's favorite food is sardines", as before, because this
# attribute remains unchanged.

bugsy._Cats__fave_food = 'counterfeit sardines'
# Only by using the mangled attribute's mangled name can we overwrite it.
# Unlike other programming languages, Python doesn't enforce read/write
# protections for private or mangled variables.
print(f"Bugsy's favorite food is {bugsy._Cats__fave_food}")
# This now prints "Bugsy's favorite food is counterfeit sardines", since we've
# forcefully overwritten the original mangled attribute
```

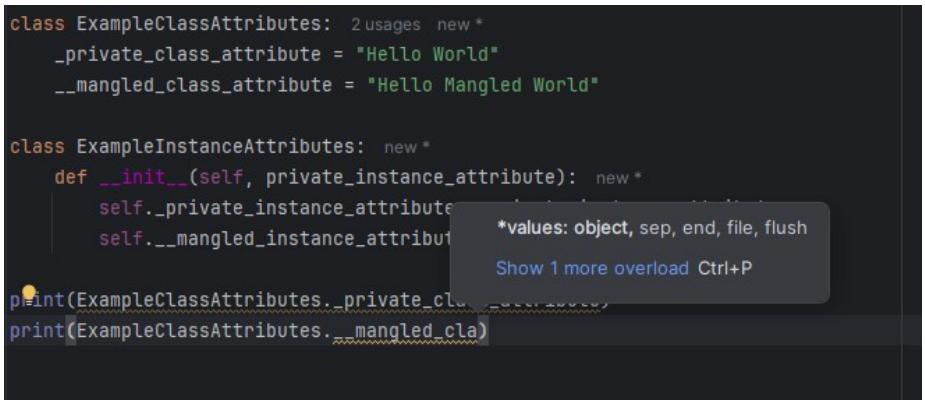

Figure 1: IDEs like PyCharm helpfully won't even show you the usual autocomplete option if you try to call a mangled attribute.

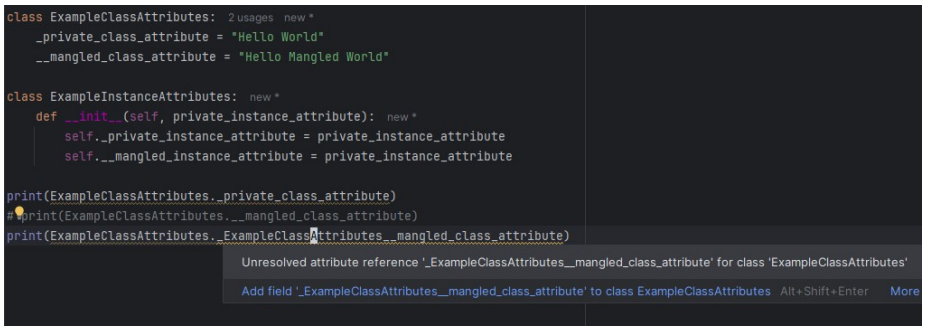

Figure 2: You can forcefully read the mangled attribute by manually typing it in. But the IDE won't autocomplete it for you, and it'll flag the attribute as an error, even though this code does print the intended string.

4 Abstraction and Encapsulation

Within computer science generally and object-oriented programming specifically, **abstraction** and **encapsulation** are fundamental concepts.

Abstraction is the concept of simplifying complexities to their most elemental level. It means hiding anything in the program that the user (be that an end-user or a programmer) doesn't need to know.

Encapsulation is the act of hiding details to achieve a satisfactory level of abstraction. In Python programming, you encapsulate data and the nitty gritty details of implementation by organizing everything into methods, attributes, and classes. This way, you and other developers don't need to know how every single thing works in order to build on the program.

Here's an analogy: When you hire movers to help you move your stuff from one apartment to another, you naturally must put all your stuff in boxes. If you're really

on top of things, you label each box so the movers know where to put it in your new place. You're *encapsulating* your tchotchkies—silverware, cat toys, heirloom bowling ball, etc.—into boxes in order to sufficiently *abstractify* the complexities of your life for your movers. In this example, you might say that `Box` is a class, and each `Box` instance has its own attributes such as `room` and `contents`. You and the movers don't need to know *how* exactly tape works in order to benefit from the `Box.tape_up()` method. That's the beauty of abstraction.

5 Properties

Properties are functions that manage attribute data. Conventionally, you create two properties for each class attribute.

Getter (Accessor) A property that retrieves data from an attribute. Indicated by prepending a function definition with `@property`. Use a getter property function to access data (and, optionally, apply formatting).

Setter (Mutator) A property that changes an attribute's data. Indicated by prepending a function definition with `@property_name.setter`, where "property_name" is the name of the function prepended with `@property`. Use a setter property function for data validation and error handling.

Simply put, the @property decorator allows us to define a method that we can access like an attribute.

Don't Forget

The **getter** (`@property`) *must* return a value that includes `self._attribute`, where `_attribute` is the private variable associated with the getter. The **setter** (`@attribute.setter`) should *not* return a value; it just redefines the private value (i.e., `self._attribute = new_attribute`).

6 Case Study: Boxes and Movers

Say we have a class called Box with two instance attributes, `contents` and `room`, each of which takes a string. Each object instance also gets its own `mover_label` attribute, which tells the mover the first item in the `contents` and the `room` where it should go. The class also has a `full_label()` method, which the user can use to read the full label of the box. The movers have been hired only to look at `mover_label`, though—they'll never read a box's full label.

```
class Box:
    def __init__(self, contents: str = "", room: str = ""):
```

```
        self.contents = contents
        self.room = room

        # Returns only the first item in contents and the name of the room
        self.mover_label = contents.split(", ")[0] + " / " + room

    def full_label(self):
        print(f"The box contains: ")
        full_list = self.contents.split(", ")
        for item in full_list:
            print(f"- {item}")
        print(f"It should go in the {self.room}.")
```

Using this class, the user can create all kinds of boxes with different contents destined for different parts of the new apartment.

```
video_games = Box(contents="Nintendo, controllers, headset", room="den")

# All the mover sees is the mover_label
print(video_games.mover_label) # Prints "Nintendo / den"

# The owner can see the full label
video_games.full_label()
"""Prints:
The box contains:
- Nintendo
- controllers
- headset
It should go in the rec room.
"""
```

But what if the user realizes, after creating the box, that they put the wrong room on it? The user can change the room by accessing the class object's attribute directly, but because the `mover_label` attribute is created when the object is instanced, it's too late—the mover will still have outdated instructions.

```
# If the owner changes the room on the box...
video_games.room = "rec room"

# Then the mover still sees the outdated instructions!
print(video_games.mover_label) # Still prints "Nintendo / den"

# The full label is changed, but that doesn't help our mover.
video_games.full_label()
"""Prints:
The box contains:
- Nintendo
- controllers
- headset
It should go in the rec room.
"""
```

What to do? One solution would be to create a new class method that writes the `mover_label` every time it's called.

```
class Box:
    def __init__(self, contents: str = "", room: str = ""):
        self.contents = contents
        self.room = room

    def mover_label(self):
        # Returns only the first item in contents and the name of the room
        return self.contents.split(", ")[0] + " / " + self.room
```

```
    def full_label(self):
        print(f"The box contains: ")
        full_list = self.contents.split(", ")
        for item in full_list:
            print(f"- {item}")
        print(f"It should go in the {self.room}.")
```

However, this means we now have to go through the code and change every instance of the `mover_label` *attribute* to the `mover_label()` *method* instead. Otherwise we get this:

```
print(video_games.mover_label)
# Now outputs: <bound method Box.mover_label of <__main__.Box object at
     0x000002AB05EF9160>>
```

. . . when what we really wanted is this:

```
print(video_games.mover_label())
# Prints the correct mover label: "Nintendo / rec room"
```

What a headache!

Other languages have getters and setters. Python has this functionality, too, and it's best accessed through **properties**. In this example, the `@property` decorator lets us define `mover_label` like a method, but we can *access* it like an attribute. All we have to do is add `@property` to the line right before we define our method:

```
    @property
    def mover_label(self):
        # Returns only the first item in contents and the name of the room
        return self.contents.split(", ")[0] + " / " + self.room
```

Now, we can still refer to the `video_games.mover_label` attribute, and it will return the return value of the `video_games.mover_label()` method. Nifty!

Let's tidy things up. A good getter returns a data value, and it shouldn't return itself. So let's change the attribute in the constructor to a private attribute (prepending it with an underscore) and have the getter method return that instead. Here's what our code looks like now.

This is all well and good. But let's say we want to write a custom label for the movers. For example, maybe we have a box of sensitive personal materials, and we'd rather not tell the mover what exactly is in there. If `mover_label` were still an attribute, we could change it directly. But because it's a method now, we can't change it like that.

```
video_games.mover_label = "Private stuff / garage"
# This results in an error: AttributeError: property 'mover_label' of 'Box'
     object has no setter
```

To do this, we need to define a **setter** in our class. This is another decorator that alters or *mutates* the attribute in question. Whereas a getter lets you access your method like an attribute, a setter lets you change your method like an attribute too. It's useful for data validation and such.

After much debugging and troubleshooting, here is the final code for a script that helps us label our boxes for movers. It also includes some logic in the room setter that validates whether room on a label matches one of the rooms in our new apartment.

```
class Box:
    def __init__(self, contents: str = "", room: str = ""):
        self._contents = contents
        self._room = room
        self._mover_label = None

    @property
    def mover_label(self):
        # Returns only the first item in contents and the name of the room
        if self._mover_label is None:
            self._mover_label = self._contents.split(", ")[0] + " / " + self._room
        return self._mover_label

    @mover_label.setter
    def mover_label(self, new_label):
        self._mover_label = new_label

    @property
    def contents(self):
        return self._contents

    @contents.setter
    def contents(self, value):
        self._contents = value

    @property
    def room(self):
        return self._room

    @room.setter
    def room(self, value):
        allowed_rooms = ['bedroom', 'bathroom', 'garage', 'dining room']
        try:
            if value not in allowed_rooms:
                raise ValueError(f"Invalid room '{value}'. "
                    "Room must be one of: {allowed_rooms}")
            self._room = value
        except ValueError as e:
            print(f"Error: {e}")

    def full_label(self):
        print(f"The box contains: ")
        full_list = self._contents.split(", ")
        for item in full_list:
            print(f"- {item}")
        print(f"It should go in the {self._room}.")


video_games = Box(contents="Nintendo, controllers, headset", room="bedroom")

print("here is the full label:")
video_games.full_label()
print("here is the mover label:")
print(video_games.mover_label)
print()


print("\nTesting invalid room assignment; changing room to rec room:")
video_games.room = "rec room"

# If the owner changes the room on the box...
print("changing room to garage...")
video_games.room = "garage"
print("changing contents...")
video_games.contents = "dirt, teacup, ears"
print()
```

```
print("new full label")
video_games.full_label()
print("new mover label: ")
print(video_games.mover_label)
# Prints the correct mover label: "Nintendo / rec room"

print("custom mover label:")
video_games.mover_label = "private things / dining room"
print(video_games.mover_label)
```

We're getting carried away. Let's continue to what else we learned in this module.

7 Inheritance

In an object-oriented programming language like Python, objects can inherit data and logic from "parent" objects. For example, everything in Python is an object, which means everything in Python effectively inherits attributes from the overarching **object class**. Furthermore, for any class, we can define a **parent class** that the class should inherit methods or attributes from. A parent class is also sometimes called a super class.

For example, let's say we have a `Pets` class to which all our pets belong. We have all kinds of pets, including cats, so we also have a `Cats` class. Every cat is a member of `Cats`, and all members of `Cats` are members of `Pets`. All pets have some of the same attributes, such as a `name`. Cats have some attributes that only make sense for cats, such as `purring_status` and `whiskers_number`. So we want a member of the `Cats` family to inherit some attributes, like `name`, but not others, such as `purring_status`. Here's how that might look in Python code.

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


class Cats(Pets):
    def __init__(self, name: str = "", is_purring: bool = False):
        self.is_purring = is_purring
        super().__init__(name=name)

    def __str__(self):
        return f"This is {self.name}, one of my cats. :3"


patches = Cats("Patches", True)
```

Organizing classes into parent and child classes is another great way to organize data in our Python programs.

8 Magic Methods

The root "object" class includes several special methods. Because all classes in Python inherit from the object class, all classes have access to these special methods. `__init__` is one of these "magic methods." Another is `__str__`. Python will give these methods

default values, but we can override those defaults by defining the methods in our class declaration.

Let's look at another example of how we might define a `Cats` class, this time modifying the built-in `__str__` method.

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


patches = Cats('Patches', 'tuxedo')
print(patches) # Prints '<__main__.Cats object at 0x73735ca340b0>'
```

If we define the `__str__` method in our class declaration, though, we can change the string to whatever we want. We can even include the name of the class instance!

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

    def __str__(self):
        return f"This is {self.name}, one of my cats. :3"


patches = Cats('Patches', 'tuxedo')
print(patches) # Prints 'This is Patches, one of my cats. :3'
```

9 More GitHub Tools

This module also covered some more ways to upload files to GitHub. Aside from using PyCharm's built-in GitHub support (a very handy feature), I covered these GitHub techniques in my knowledge document for module 05.

10 Creating the Program

The program we created for this module's assignment is the same program as the previous module, except now it makes use of classes and the fundamental concepts of object-oriented programming.

For all that we covered in this module, the assignment was pretty straightforward. It mainly consisted of converting data to objects, in accordance with the standards of object-oriented programming. One of the changes was to convert this:

```
# Convert the list of dictionary rows into a list of Student objects
student_objects = []
# TODO replace this line of code to convert dictionary data to Student data
student_objects = json_students
```

to this:

```
# Convert the list of dictionary rows into a list of Student objects
student_objects = []
for row in json_students:
    student_obj = Student(first_name=row["FirstName"],
```

```
                        last_name=row["LastName"],
                        course_name=row["CourseName"])
    student_objects.append(student_obj)
```

This lets the program access student data as object data rather than dictionary rows. Each student is its own class object, rather than a dictionary object. This means that instead of accessing a student's first name via the `"FirstName"` key, we'll access the same information via its `first_name` attribute.

Here's how that looks, later on in the program.

Before:

```
for student in student_data:
    print(f'Student {student["FirstName"]} '
          f'{student["LastName"]} is enrolled in {student["CourseName"]}')
```

After:

```
for student in student_data:
    print(f'Student {student.first_name} {student.last_name} '
          f'is enrolled in {student.course_name}')
```

And later still, in our `input_student_data` method, we can write new input from the user directly to a new `Student` object, rather than a dictionary row.

Before:

```
student = {"FirstName": student_first_name,
           "LastName": student_last_name,
           "CourseName": course_name}
```

After:

```
student = Student(first_name=student_first_name,
                  last_name=student_last_name,
                  course_name=course_name)
```

10.1 Debugging

After finishing the extant to-do items in the assignment starter file, I tested the program but was getting errors. Every time I chose option 3 to save student data to the JSON file, the program would throw me an error and crash. Furthermore, the Enrollments file would be erased. Interesting behavior! Here's a readout of what my terminal produced:

```
C:\Users\Patrick\AppData\Local\Programs\Python\Python313\python.exe
    C:\..._Module07\Assignment\A07\Assignment07.py
Error: There was a problem with reading the file.

-- Technical Error Message --
Expecting value: line 1 column 2 (char 1)
Subclass of ValueError with the following additional properties:

msg: The unformatted error message
doc: The JSON document being parsed
pos: The start index of doc where parsing failed
lineno: The line corresponding to pos
colno: The column corresponding to pos
```

```
<class 'json.decoder.JSONDecodeError'>
Traceback (most recent call last):
  File "C:\..._Module07\Assignment\A07\Assignment07.py", line 284, in <module>
    students = FileProcessor.read_data_from_file(file_name=FILE_NAME)
  File "C:\..._Module07\Assignment\A07\Assignment07.py", line 126, in
     read_data_from_file
    return student_objects
           ^^^^^^^^^^^^^^^
UnboundLocalError: cannot access local variable 'student_objects' where it is not
     associated with a value

Process finished with exit code 1
```

To resolve this, I examined the `write_data_to_file` method. There was a `# TODO` comment there, but it was marked as (Done). Maybe an error?

```
# TODO Add code to convert Student objects into dictionaries (Done)

file = open(file_name, "w")
json.dump(student_data, file)
file.close()
IO.output_student_and_course_names(student_data=student_data)
```

This didn't look quite right. I made sure that the code was creating a suitable dictionary for the JSON file, and I made sure that dictionary object was populated with correctly formatted student information from the `student_data` variable (which was now, remember, a list of `Student` object instances, not dictionary rows). I used a helper variable, `json_students`, to achieve this, and built out a basic `for` loop to populate the helper variable. Finally, I changed the first parameter of the `json.dump()` method to the helper variable containing the correctly formatted list of dictionary rows. (I also added the optional `indent` parameter to make the JSON file a bit prettier.) Here is the result:

```
json_students = []
for student in student_data:
    student_dict = {"FirstName": student.first_name,
                    "LastName": student.last_name,
                    "CourseName": student.course_name}
    json_students.append(student_dict)

file = open(file_name, "w")
json.dump(json_students, file, indent=2)
file.close()
IO.output_student_and_course_names(student_data=student_data)
```

With this done, the program now worked as intended.

11 Conclusion

In this module, we covered a ton of stuff! We learned more about classes and how to use them—along with constructors, private and mangled variables, and getter and setter decorators—to better organize our code in accordance with encapsulation and abstraction best practices and to protect data in our program from errant manipulation. We also touched on Python's magic methods (like `__int__()` and `__str__()`), inherited classes, and how to convert JSON data to custom objects and vice-versa.

Here is a link to this knowledge document and the accompanying script file: https://github.com/reniepUW/IntroToProg-Python-Mod07