Sam Hollister

6/4/25

IT FDN 110 A

Assignment 07

https://github.com/samhollister/IntroToProg-Python-Mod07

Classes and Objects

Introduction

This week, we expanded on the concept of classes to include objects in addition to functions. Creating objects from classes allows us to assign unique characteristics, such as constructors, private attributes, and getter/setter properties. We also covered parent-child relationships and inheritance between classes.

Using data classes makes the code more complex, but these tools are important to the concepts of abstraction and encapsulation. It keeps the main body of the script relatively simple, by hiding steps such as data validation and error handling within the class set up. Private attributes also protect data from users who may be less familiar with the intricacies of the program: by abstracting the way in which users interact with the data, it creates code that is less error prone.

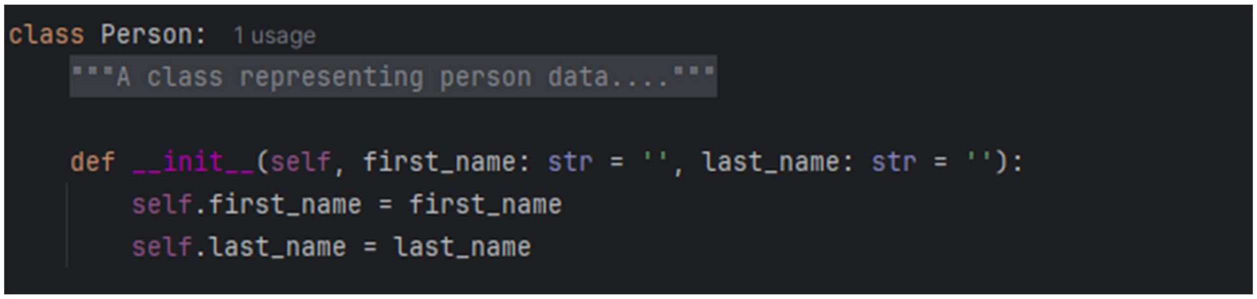
Objective

The objective of this week's assignment is to use what we learned about classes and objects to modify the same general script we have worked with in previous weeks: a program to input, output, and write student course registration data.

To do this, we need to define new classes "Person" and "Student", modify the code to convert from dictionaries to objects and vice versa, and update our I/O functions to use our new object types. The result should be a program that functions identically to last week from a user perspective but is more maintainable and structured.

Defining Object Classes

This assignment stipulates that we should create two classes: "Person" and "Student". These two classes should have a parent-child relationship such that "Student" inherits from "Person". Thus, I started by defining the "Person" class. The first step is adding DocString notes, and the constructor method. The constructor method includes two string parameters: first_name and last_name.

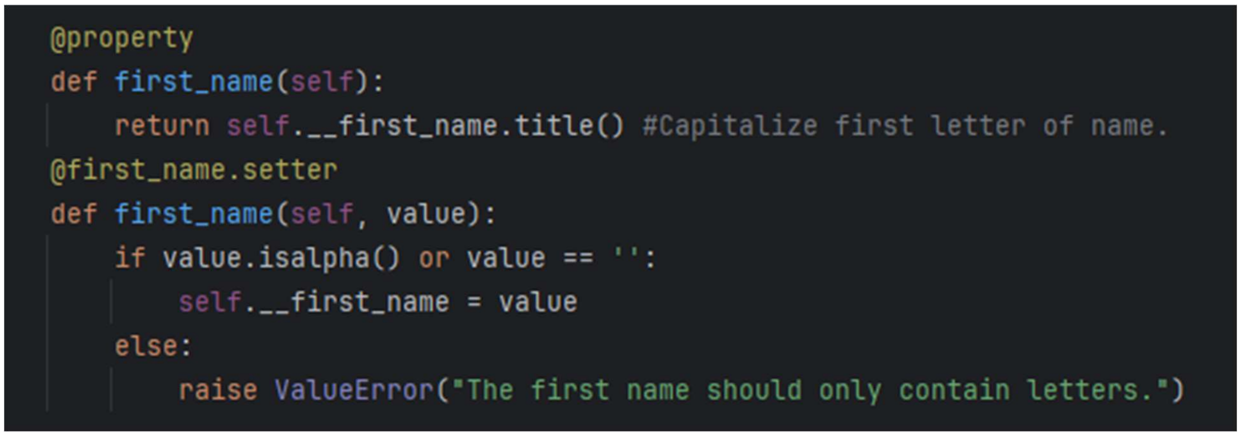


```python
class Person:  1 usage
    """A class representing person data...."""

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

Figure 1.1: The "Person" class and constructor

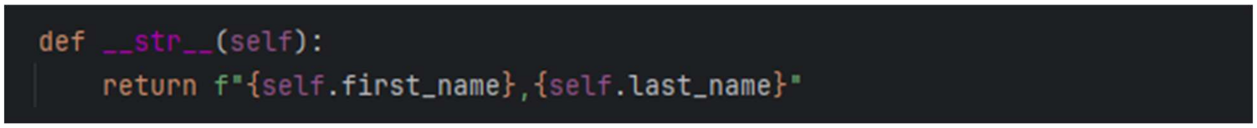
Next, I added getter and setter properties for the first_name parameter. In the getter property, I define it to return the __first_name private attribute, with the .title() method applied to capitalize the first letter of the name. This creates a layer of abstraction: when users call Person.first_name, they will not be able to call the underlying __first_name private attribute. For the setter property, I add a data validation step to ensure that the passed value is alphabetical. If it is not, it will raise a value error with a custom message.



```python
    @property
    def first_name(self):
        return self.__first_name.title() #Capitalize first letter of name.
    @first_name.setter
    def first_name(self, value):
        if value.isalpha() or value == '':
            self.__first_name = value
        else:
            raise ValueError("The first name should only contain letters.")
```

Figure 1.2: Getter/setter properties for first_name

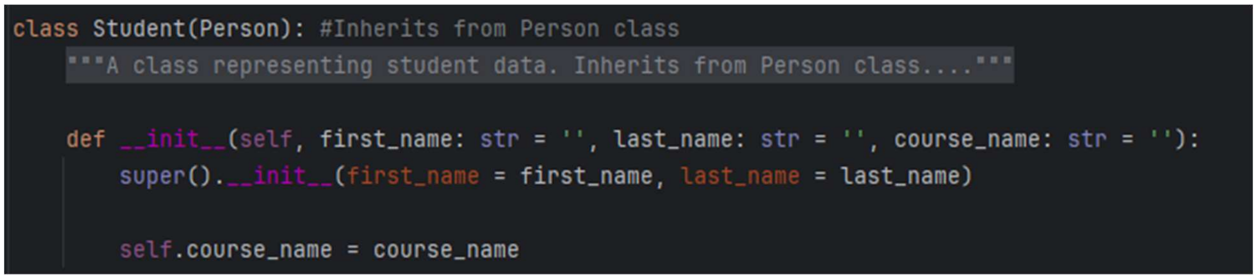
Next, I repeated this same process for the last_name parameter. The last component of the "Person" class was to override the __str__() "magic" method, to return a comma-separated list of parameter values rather than the object location in memory.



```python
    def __str__(self):
        return f"{self.first_name},{self.last_name}"
```

Figure 1.3: Overriding the __str__() method

Next, I created the "Student" class. Like the "Person" class, I started with the DocString notes and constructor. This time, I added additional code to explicitly state the inheritance relationship. It invokes the properties of the "Person" object for the first_name and last_name parameters, which leaves only the course_name parameter to define.



```python
class Student(Person): #Inherits from Person class
    """A class representing student data. Inherits from Person class...."""

    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
```

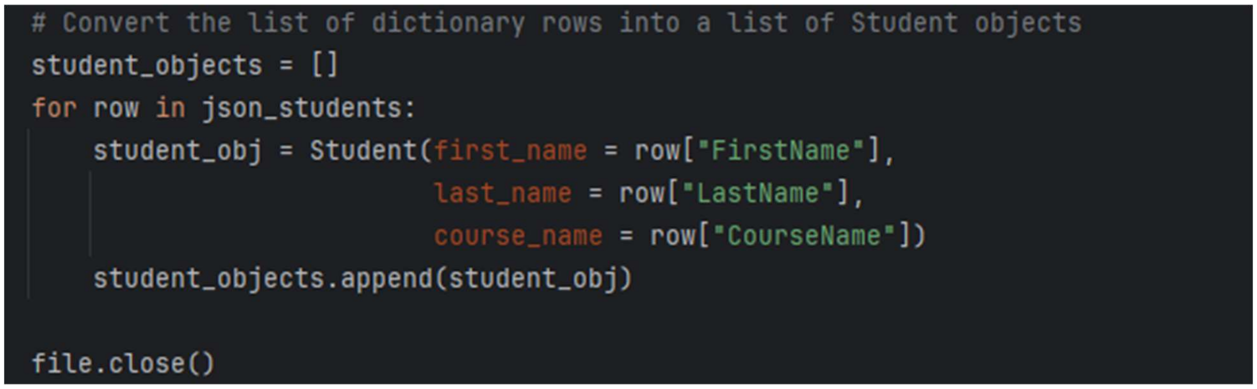
Figure 1.4: Defining the "Student" class, which inherits from "Person"

Finally, I repeated the step to override the __str__() method to instead return a comma-separated list of values. With the new classes defined, the next step was to update the script to interact with the Student and Person objects.

Updating Processing

In the existing FileProcessor class, there are two defined functions: read_data_from_file() and write_data_to_file(). Because the objective of these functions is read from and write to a JSON file, the script needed to be updated to convert the JSON data into a list of Student objects, and then to convert that list of objects back into JSON format.

First, I modified the read_data_from_file() function. The new code uses a for loop to convert each row of the loaded JSON data into a separate Student object, which is then appended to a new list "student_objects".

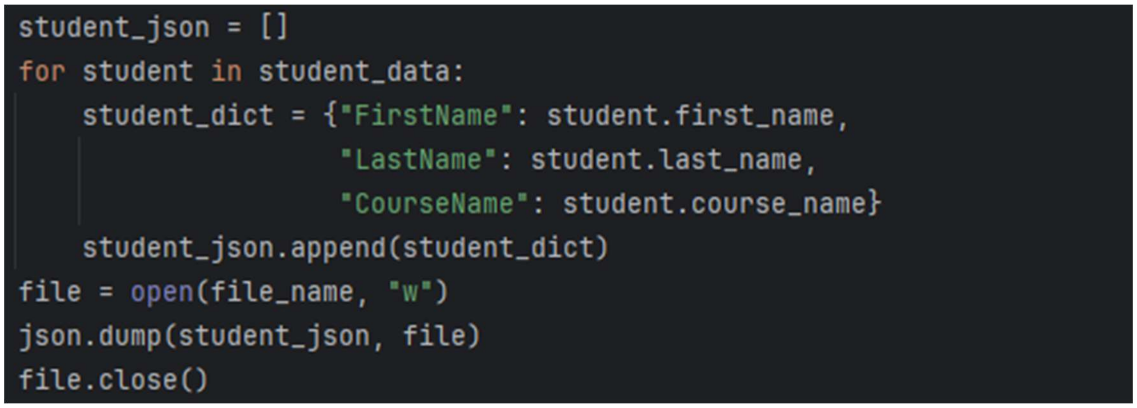


```python
# Convert the list of dictionary rows into a list of Student objects
student_objects = []
for row in json_students:
    student_obj = Student(first_name = row["FirstName"],
                          last_name = row["LastName"],
                          course_name = row["CourseName"])
    student_objects.append(student_obj)

file.close()
```

Figure 2.1: Converting JSON data to a list of objects

Next, I updated the write_data_to_file() function to convert the list of objects back into JSON format. This also uses a for loop to convert each object in the list into a dictionary, which is appended to a new list named "student_json". That new list is then written to the JSON file.



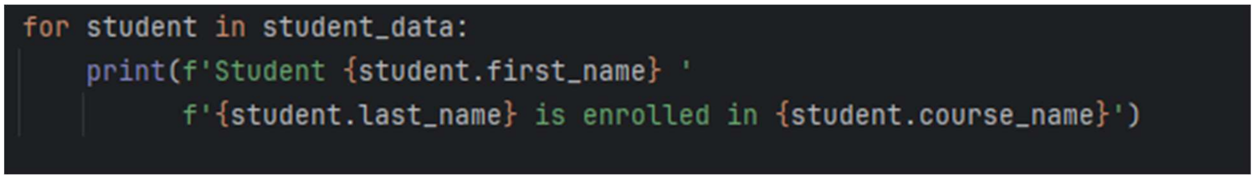
```
student_json = []
for student in student_data:
    student_dict = {"FirstName": student.first_name,
                    "LastName": student.last_name,
                    "CourseName": student.course_name}
    student_json.append(student_dict)
file = open(file_name, "w")
json.dump(student_json, file)
file.close()
```

Figure 2.2: Converting a list of objects to JSON format

Updating Presentation

In the IO (input/output) class, there are also two functions which required updates to reference the new Student object: output_student_and_course_names() and input_student_data().

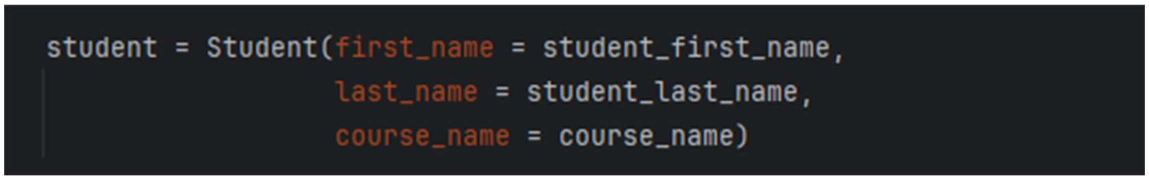
In the output function, the change was simple. I updated the print statement to include references to the Student object instead of a dictionary object.



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

Figure 3.1: Updated output function

In the input function, I changed the code so that the "student" object would be a Student type rather than a dictionary.



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

Figure 3.2: Updated output function

Thanks to the modularity of the code, these were the only required IO code changes.

Summary

At the beginning of this module, I had a hard time understanding the benefits of using classes to define objects. It seemed like it added a great deal of complexity to the code. As I got used to using them, I began to understand the ways in which is makes code more reliable, and can reduce complexity elsewhere in the script. For example, putting the data validation steps in the class definition improves the readability of the code in other areas. The more that can be done or separated into smaller chunks, the easier it is to understand the main script body.

It felt odd at first to be updating references throughout the script. But as I understood the concepts of abstraction and encapsulation more, it made more sense as to why it was desirable. In my day job, it can be very disruptive when someone makes changes to a table or spreadsheet directly. It may break downstream references or cause unintended changes. Controlling which data a user can directly edit makes code less susceptible to unintentional error. I learned a lot this week, and am looking forward to applying everything in the final assignment!