Shannon Brown

February 27, 2024

IT FDN 110 A

Assignment 7

https://github.com/sdbrown8/IntroToProg-Python-Mod07

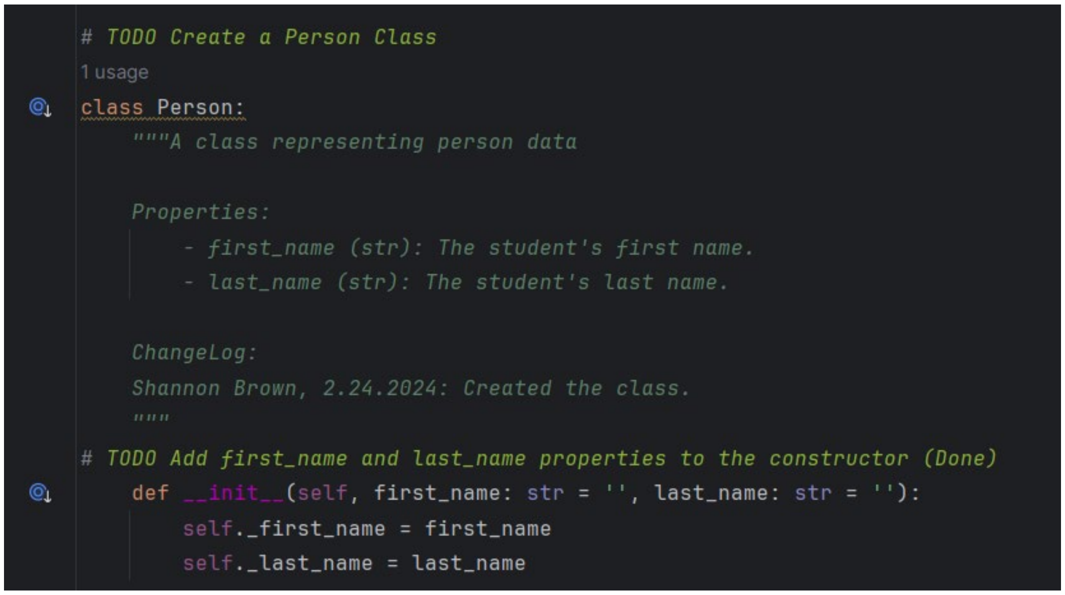
Classes and Objects

Introduction

We continued to hone in on the use of statements, functions, and classes this week. The difference between data class, presentation class, and processing class were introduced, and these tools are employed to create concise code and reduce repetitiveness. We reviewed video modules, readings, and labs. The purpose of this paper is to outline how we expanded on our previous script from Assignment06 using data classes.

Creating a Python Script

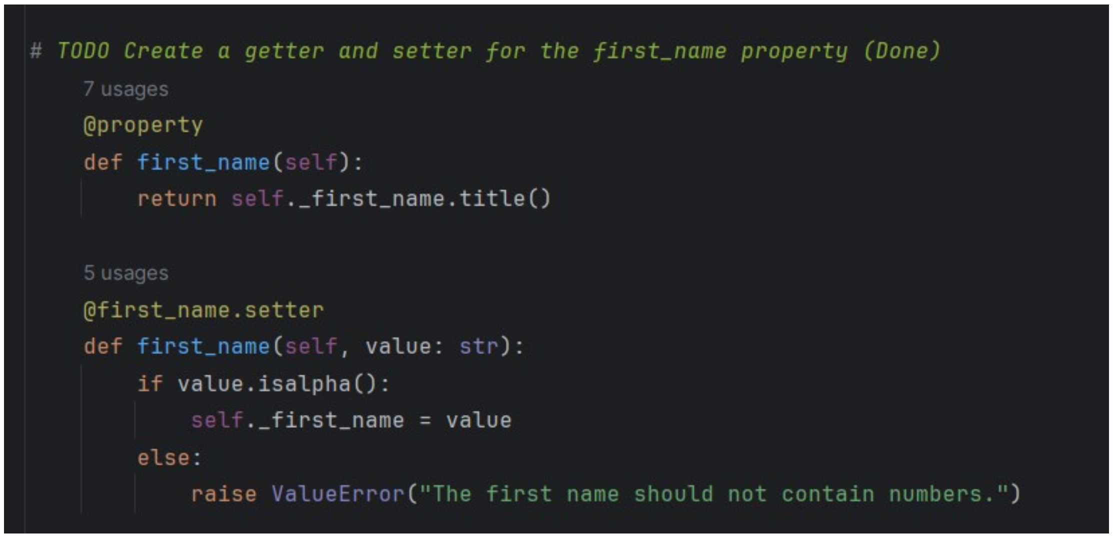
The aim of Assignment07 was to expand on our script from Assignment06 with data classes and objects. The overall goal of allowing the user to select from a menu, input student data, view the data, and store it in a JSON file still remained. To begin, I opened Assignment07-Starter.py in PyCharm IDE, updated the header, and resaved the file as Assignment07.py. I then checked that the constants and variables met the acceptance criteria. Next, I created the Person class, added descriptive document string, and I defined a constructor. The variables were defined as strings within the constructor, and the self keyword was used to refer to data found not directly in the class (Figure 1).



```
# TODO Create a Person Class
1 usage
class Person:
    """A class representing person data

    Properties:
        - first_name (str): The student's first name.
        - last_name (str): The student's last name.

    ChangeLog:
    Shannon Brown, 2.24.2024: Created the class.
    """
# TODO Add first_name and last_name properties to the constructor (Done)
    def __init__(self, first_name: str = '', last_name: str = ''):
        self._first_name = first_name
        self._last_name = last_name
```

Figure 1. Creation of the Person class and the defining of a constructor

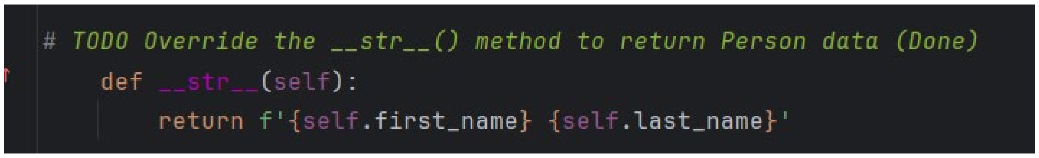
For the Person class, since I knew I would need functions to access and modify attributes, I began by using the @property decorator to collect data. In order to add validation and error handling, I used the @setter decorator. The value.isalpha() function was used for error handling and to prevent input of names that were not entirely composed of letters (Figure 2). The same process was repeated for the last_name.



```
# TODO Create a getter and setter for the first_name property (Done)
    7 usages
    @property
    def first_name(self):
        return self._first_name.title()

    5 usages
    @first_name.setter
    def first_name(self, value: str):
        if value.isalpha():
            self._first_name = value
        else:
            raise ValueError("The first name should not contain numbers.")
```

Figure 2. Use of property functions to manage attribute data

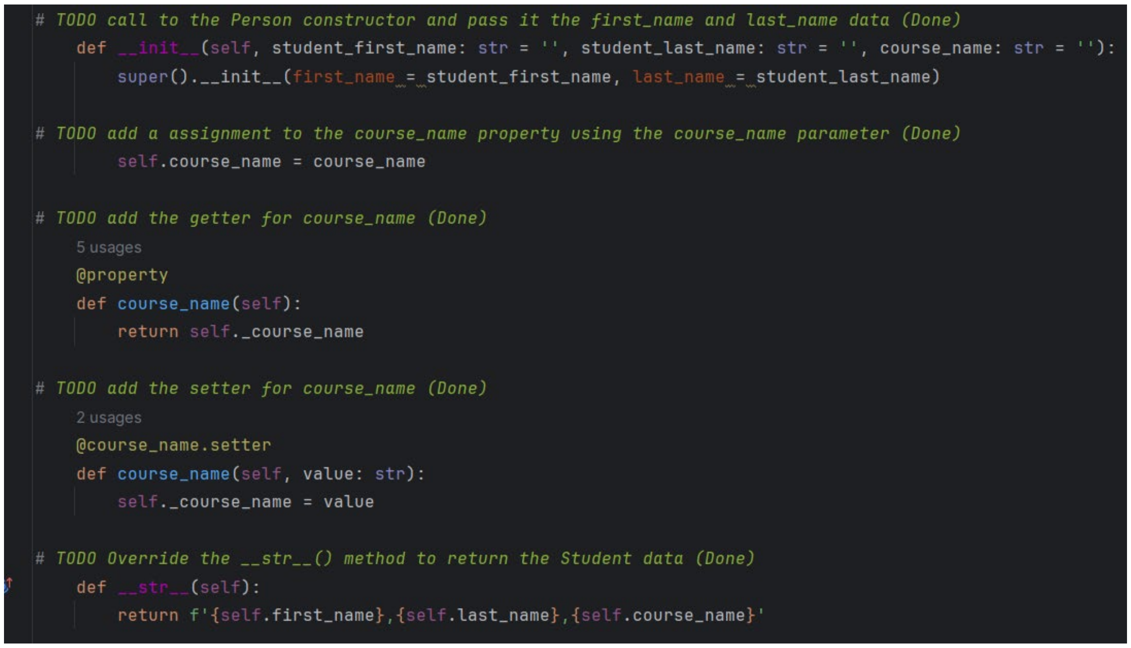
By using magic methods in the script, specifically __str__, I was able to control the string representation of the object from Person class so it was more user-friendly (Figure 3).



```
# TODO Override the __str__() method to return Person data (Done)
    def __str__(self):
        return f'{self.first_name} {self.last_name}'
```

Figure 3. Used the string method to define the representation of the object

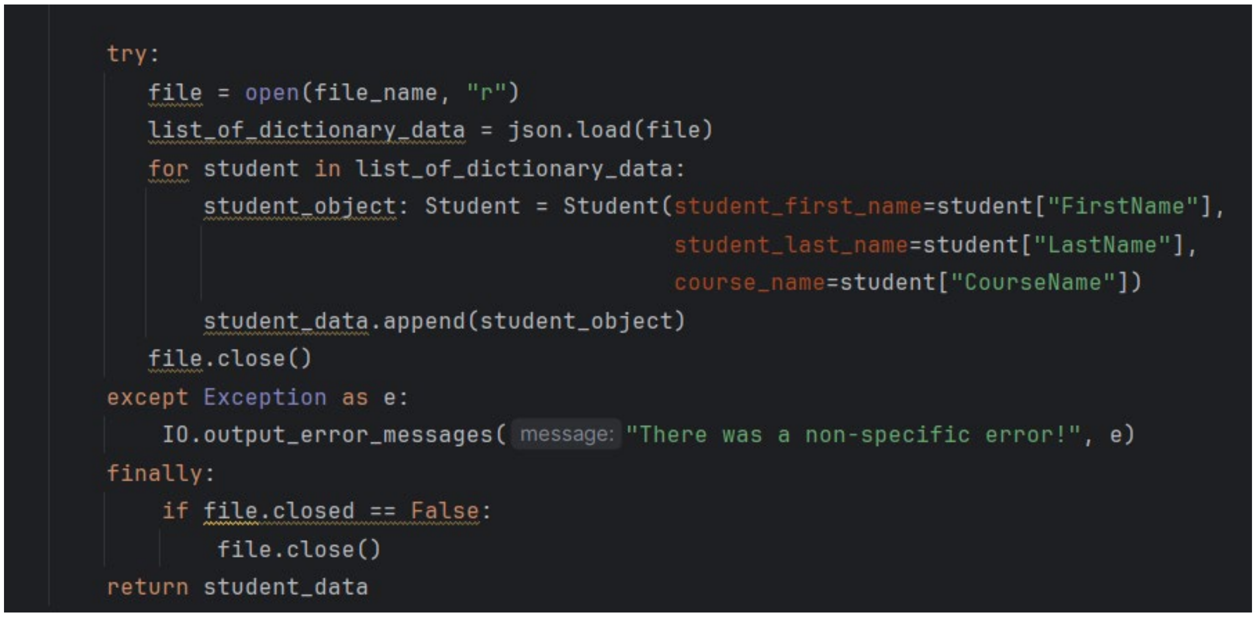
The next step was to create a Student class which inherits data from the Person class. I started by adding a class name to indicate explicit inheritance (i.e., class Student(Person)). Then I added a descriptive document string and defining a __int__ constructor. The super() function in front of the constructor (__int__) was used to call the constructor of the parent Person class, which initialized the first_name and last_name attributes (Figure 4). Since the course_name was unique to the Student class, I called that as a unique attribute. Then, as I had previously done with the names, I used properties functions to get and set the data. Lastly, in this class, I used __str__() to override the default behavior from the parent class Person and return a coma-separated string of the Student class's data (Figure 4).



```
# TODO call to the Person constructor and pass it the first_name and last_name data (Done)
    def __init__(self, student_first_name: str = '', student_last_name: str = '', course_name: str = ''):
        super().__init__(first_name = student_first_name, last_name = student_last_name)

# TODO add a assignment to the course_name property using the course_name parameter (Done)
        self.course_name = course_name

# TODO add the getter for course_name (Done)
    5 usages
    @property
    def course_name(self):
        return self._course_name

# TODO add the setter for course_name (Done)
    2 usages
    @course_name.setter
    def course_name(self, value: str):
        self._course_name = value

# TODO Override the __str__() method to return the Student data (Done)
    def __str__(self):
        return f'{self.first_name},{self.last_name},{self.course_name}'
```

Figure 4. Creating a Student class that inherits data from class Person

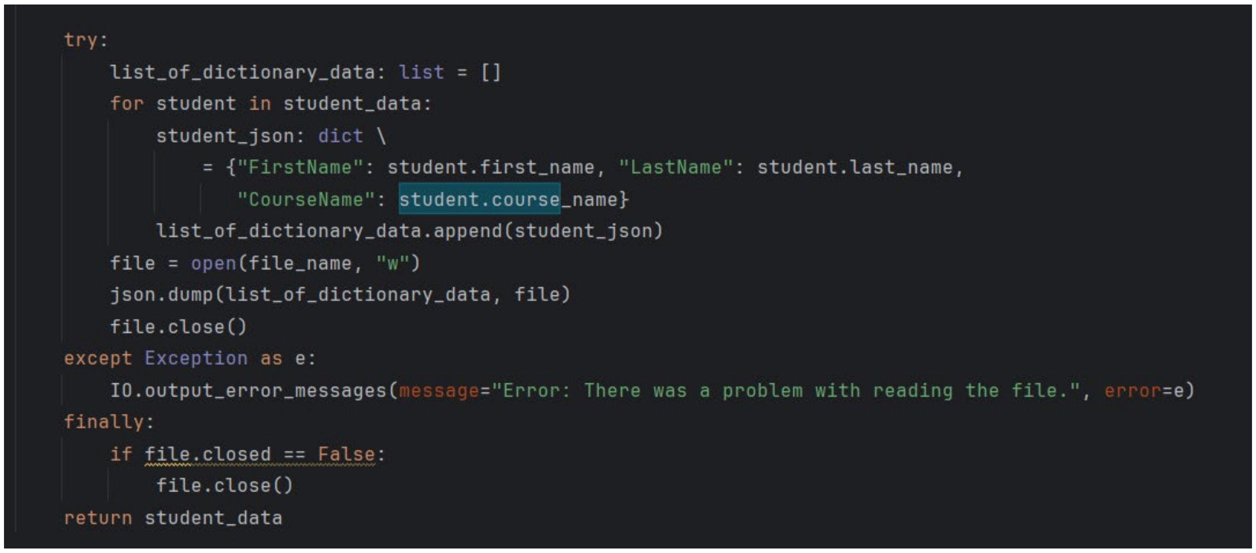
After troubleshooting my classes and using breaks and debugging to ensure they properly function, I began reviewing the remainder of the code. In the read_data_from_file() function, as outlined in Mod07-Lab01-WorkingWithConstructors, I changed the variable names so that the data read as a list of dictionary rows. Then I converted the list of dictionary data in a list of student objects which was appended to student_data (Figure 5).



```
try:
    file = open(file_name, "r")
    list_of_dictionary_data = json.load(file)
    for student in list_of_dictionary_data:
        student_object: Student = Student(student_first_name=student["FirstName"],
                                          student_last_name=student["LastName"],
                                          course_name=student["CourseName"])
        student_data.append(student_object)
    file.close()
except Exception as e:
    IO.output_error_messages( message: "There was a non-specific error!", e)
finally:
    if file.closed == False:
        file.close()
return student_data
```

Figure 5. Updating the read_data_from_file() function to use list of dictionaries

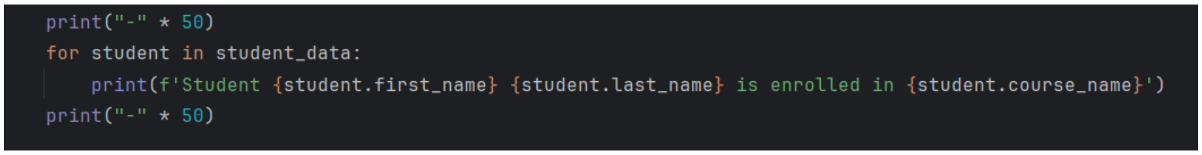
Next, relying on the script from the same module, Mod07-Lab01-WorkingWithConstructors, I located the write_data_to_file() method and created a new variable called list_of_dictionary_data. Code was written to convert the list of Student objects to a JSON compatible list of dictionaries. Then I simply changed the first argument for my json.dump() function to this dictionary variable (Figure 6).



```
try:
    list_of_dictionary_data: list = []
    for student in student_data:
        student_json: dict \
            = {"FirstName": student.first_name, "LastName": student.last_name,
               "CourseName": student.course_name}
        list_of_dictionary_data.append(student_json)
    file = open(file_name, "w")
    json.dump(list_of_dictionary_data, file)
    file.close()
except Exception as e:
    IO.output_error_messages(message="Error: There was a problem with reading the file.", error=e)
finally:
    if file.closed == False:
        file.close()
return student_data
```

Figure 6. Updating the write_data_to_file() function to use list of dictionaries

I continued to review the document and make additional updates. For instance, under class IO, in the output_student_and_course_names() function, I switched from using dictionary keys to using a list of student object's attributes (Figure 7). A similar strategy was utilized when updating the print() function associated with the input_student_data function under class IO. However, in addition, excess text, such as error handling which was now included in the Person class was removed to make less redundant.



```
print("-" * 50)
for student in student_data:
    print(f'Student {student.first_name} {student.last_name} is enrolled in {student.course_name}')
print("-" * 50)
```

Figure 7. Converted from using list of dictionary data keys to a list of student object attribute

Confirming Functionality of the Script

On PyCharm, I right clicked and then selected "Run Assignment07" to test my code and utilized debugging tools to fix any small errors, including missing syntax (e.g., quotes, parentheticals). Once the code was tested on PyCharm, I opened the command prompt to test the program in the terminal. To use python to run my created program, I typed "python.exe" followed by the Assignment07.py, after relocating to the program's home directory. I confirmed the code was run correctly on the terminal, checked the JSON file for accuracy, and ensured the script met all parameters set by the acceptance criteria.

Summary

Assignment 07 employed previous programming tools such as dictionaries, lists, and error handling; however, we expanded on this work using different classes to create more concise, user-friendly code. All information required for the assignment was available via the module videos, readings, or labs. These tools will be further employed when I work with larger data sets in the future.