Edwin Kintu-Lubowa

November 24, 2024

Foundations of Programming: Python

Assignment07

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

Classes, Objects and Inheritance

Introduction

This week's assignment was to create a program that uses constants, variables, and print statements to display a message about a student's registration. The assignment is very similar to Assignment06, however, it introduces two data classes: Person and Student to handle the program data. In this assignment, the focus was, still, centered around the pattern of 'Separation of Concerns' and in particular, the Data layer. In this assignment, I learned how to create a Data layer that included two classes and the methods to handle the data. In this knowledge document, I will chronicle the concepts that I learned, and how they aided and guided me to the successful completion of this assignment.

Writing the program

I started the assignment by loading the starter file into PyCharm, updated the file header with my names and the date, then renamed and saved the file. The starter file had specific "TODO" comments that offered a structured and orderly guide for where to write the code.

The first thing I did was to run the code in the starter file before I made any modifications to the file, I entered some data and double checked the 'Enrollments.json' file to ensure that the program was running as expected, and it was. The data in the .JSON file was saved as a comma separated list of dictionaries.

A turning point

In this assignment, the challenge was to create student objects from the Data classes to work with. In the prior assignments, I was working, first: with string data; then next I learned how to save formatted data to CSV files; followed by using looping and conditional logic to control the program flow; later on, I learned how to process data using lists and files, followed by Advanced Collections and Error handling where I learned how to work with lists and dictionaries, and the JSON module; and then, last week, the assignment was centered around Functions, classes and the 'Separation of Concerns' pattern — and until this point, I had

primarily dealt with data in form of collections of lists and dictionaries. This week marks a major turning point towards more advanced programming methods and concepts.

The Data Layer Classes

The Person Class

I created two classes: Person and Student, with Person as the superclass and Student as the subclass. For each class I included document strings and then created a constructor, first, for the Person class with private attributes: first_name and last_name, followed by the property methods. Below is a screen shot of the Person class definition including the document string, the constructor, and the property methods. (Figure 1)

```
29
30  class Person:  1 usage
31      """
32      A class representing person data
33
34      Properties:
35      - first_name (str): The student's first name
36      - last_name (str): The student's last name
37
38      ChangeLog: (Who, When, What)
39      Edwin Kintu-Lubowa,11/23/2024,Created the class.
40      """
41
42      # Create a constructor with private attributes for the first_name and last_name data
43      def __init__(self, first_name: str = "", last_name: str = ""):
44          self.first_name = first_name
45          self.last_name = last_name
46
```

Figure 1: Person class definition alongside the document string and constructor

After creating the constructor for the Person class, I created the property method for the private attributes ‘first_name’ and ‘last_name’. The property getter method has some formatting using .title() to capitalize the first letter of each name whilst the rest of the letters are displayed in lower case. The setter method has includes some error checking for the name values — allowing for only alphabetic characters — and also allows for an empty string. After the property methods for the private attributes, the overrides the __str__() method with first name and last names.

“The __str__ method in Python is responsible for returning a human-readable string representation of an object. As we have seen, by default, the__str__() method in your class, Python's default implementation

of __str__() will return a string that includes the object's memory address." (Randal Root: Introduction to Programming with Python: Module 07 - Classes and Objects: Page 22 - Fall 2024)

Below is a screenshot of the property methods — the getter and the setter — for the private attributes "first_name" and "last_name" for the Person class, followed by the __str__() 'to string' method, with override code, that displays the student first and last names. (Figure 2)

```
30      class Person:  1 usage
46
47          # Create property getter and setter for first name using the same code as in the Student class
48          @property  # (Use this decorator for the getter or accessor)  10 usages (8 dynamic)
49          def first_name(self):
50              return self.__first_name.title()   # formatting code
51
52          @first_name.setter  # (use this decorator for the setter or mutator)  9 usages (8 dynamic)
53          def first_name(self, value: str):
54              if value.isalpha() or value == "":
55                  self.__first_name = value
56              else:
57                  raise ValueError("First name should not contain numbers!")
58
59          # Create property getter and setter for last name using the same code as in the Student class
60          @property  # (Use this decorator for the getter or accessor)  10 usages (8 dynamic)
61          def last_name(self):
62              return self.__last_name.title()   # formatting code
63
64          @last_name.setter  # (use this decorator for the setter or mutator)  9 usages (8 dynamic)
65          def last_name(self, value: str):
66              if value.isalpha() or value == "":
67                  self.__last_name = value
68              else:
69                  raise ValueError("Last name should not contain numbers!")
70
71          # Add code to inherit code from the person class
72          def __str__(self):
73              return f"{self.first_name},{self.last_name}"
74
```

Figure 2 - The getter, the setter, and __str__() methods

The Student Class

The second class I created in the Data Layer was the 'Student' class that inherited methods and propertied from the Person class. For the Student class, I created the constructor with three private attributes, two of which were inherited from the Person class. The constructor for the Student class was immediately followed by 'super()' method that is used to access the methods and properties of the parent class — Person. The Student class has an extra private

attribute ‘course_name’. I created another pair of getter and setter property methods to process the ‘course_name’ private attribute whilst the code uses the super() method to access the other two private attributes ‘first_name’ and ‘last_name’ from the Person class. Below is code showing the class definition, the document string, followed by the constructor for three private attributes.

Below the constructor, is the super() method that is used to access the properties and methods for the Person class. There’s a pair of property functions to process the ‘course_name’ private attribute, and finally the __str__() ‘to string’ method is used to override the default string for the print statement with three private attributes for the Student object: ‘first_name’, ‘last_name’, and ‘course_name’. (Figure 3)

```
class Student(Person):  3 usages
        A class representing student data

        Properties:
         first_name (str): The student's first name.
         last_name (str): The student's last name.
         course_name (str): course name student's enrolled in

        ChangeLog: (Who, When, What)
        Edwin Kintu-Lubowa,11/23/2024,Created Class
        Edwin Kintu-Lubowa,11/23/2024,Added properties and private attributes
        Edwin Kintu-Lubowa,11/23/2024,Moved first_name and last_name into parent class
        """

    # Create a constructor with private attributes for the first_name, last_name ,and course name data
    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

    @property  # (Use this decorator for the getter or accessor)  5 usages (2 dynamic)
    def course_name(self):
        return self.__course_name.title()   # formatting code

    @course_name.setter  # (use this decorator for the setter or mutator)  4 usages (2 dynamic)
    def course_name(self, value: str):
        self.__course_name = value

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

Figure 3 - the Student class method and properties

Inheritance

The Student class, being the subclass, inherited properties and methods form the Person class, which is the superclass.

“Inheritance is a core concept in OOP where a new class (child class) can inherit data and behaviors (properties and methods) from an existing class (parent class). "Inherited code" refers to code that is inherited or derived from another source, such as a parent class or codebase from a previously developed project. Inherited code serves as the foundation for building new code or extending existing functionality.” (Randal Root: Introduction to Programming with Python: Module 07 - Classes and Objects: Page 21 - Fall 2024)

The Errors

Upon completion of the coding for the data layer, I run the program, it displayed the menu of choices: when I selected Option 2 - Show Current Data, the program displayed the data from the JSON file. However, when I entered new student data and tried to display it, the program threw an error message. I am trying to use indexing on an object that is a function, perhaps?! So, I had to go back to Lab03 and revisit how the FileProcessor methods were processing Student objects. Below is the screen shot of the error. (Figure 4)

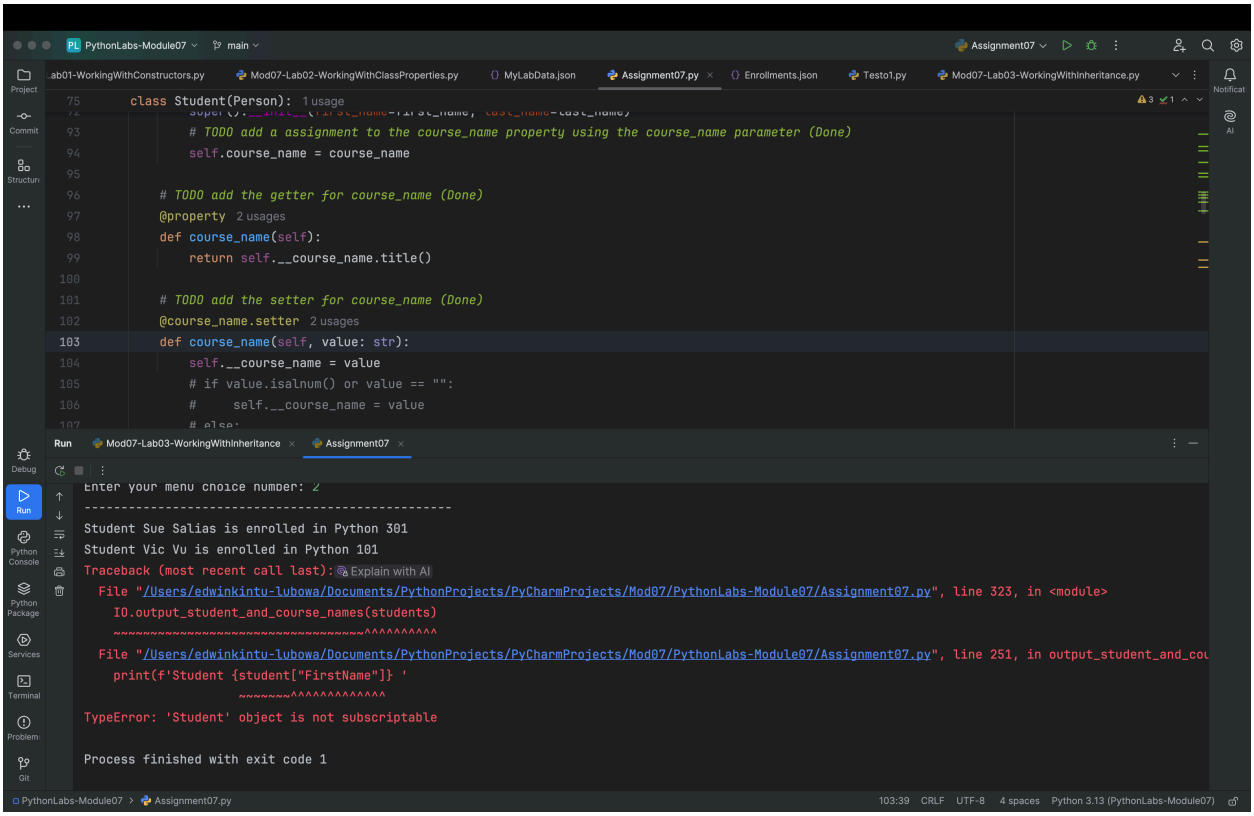

Figure 4 - error trying to process student object using ‘Display Current Data’

Rewriting FileProcessor methods

After reviewing Lab03, I had to rewrite read_from_file() and write_to_file() methods. In the read_from_file() method, I defined a list attribute ‘list_of_dictionary_data’ to receive a list of dictionary rows from the json file. Then, iterated through the list using a for loop and converting each dictionary row into an object ‘student_object’ and then appending it to the list table ‘student-data’. With the new code running, I commented out the old code from the starter file. The code still has the same lines of error handling as did the old code. Below is a screen shot of the updated code. (Figure 5)

```
110     class FileProcessor:  2 usages
119         def read_data_from_file(file_name: str, student_data: list):
133             file = open(file_name, "r")
134
135             list_of_dictionary_data = json.load(file)  # the load function returns a list of dictionary rows.
136             for student in list_of_dictionary_data:
137                 student_object: Student = Student(first_name=student["FirstName"],
138                                                   last_name=student["LastName"],
139                                                   course_name=student["CourseName"])
140                 student_data.append(student_object)
141
142             file.close()
143         except FileNotFoundError as e:
144             IO.output_error_messages( message: "Text file must exist before running this script!", e)
145         except Exception as e:
146             IO.output_error_messages( message: "There was a non-specific error!", e)
147         finally:
148             if not file.closed:
149                 file.close()
150         return student_data
151
152         # I will not be using the code below; it does not handle student objects
153         # try:
154         #     file = open(file_name, "r")
155         #     student_data = json.load(file)
156         #     file.close()
157         # except Exception as e:
158         #     IO.output_error_messages(message="Error: There was a problem with reading the file.", error=e)
159         # finally:
160         #     if file.closed == False:
```

Figure 5 - Code converting list of dictionaries into list of student objects

Conversely, I needed to convert the list of student objects into a list of dictionary rows before it could be written into a json file. Again, I consulted with Lab03 to learn how this was attained. In this scenario, again I defined a list attribute ‘list_of_dictionary_data’ that was set to empty. I then iterated through list ‘student_data’ and converted each student object into a dictionary row using the ‘key:value’ pairing and accessing the object data for the value bit of the pair and then appending the each row into the list attribute ‘list_of_dictionary_data’ The screen shot below shows the code that converted the student objects into a list of dictionaries (Figure 6)

```
110     class FileProcessor:  2 usages
165         def write_data_to_file(file_name: str, student_data: list):
178             try:
179                 list_of_dictionary_data: list = []
180                 for student in student_data:  # Convert List of Student objects to list of dictionary rows.
181                     student_json: dict \
182                         = {"FirstName": student.first_name, "LastName": student.last_name, "CourseName": student.course_name}
183                     list_of_dictionary_data.append(student_json)
184
185                 file = open(file_name, "w")
186                 json.dump(list_of_dictionary_data, file)
187                 file.close()
188             except TypeError as e:
189                 IO.output_error_messages( message: "Please check that the data is a valid JSON format", e)
190             except Exception as e:
191                 IO.output_error_messages( message: "There was a non-specific error!", e)
192             finally:
193                 if not file.closed:
194                     file.close()
195
196             # I will not be using the code below; it does not handle student objects
197             # try:
198             #     file = open(file_name, "w")
199             #     json.dump(student_data, file)
200             #     file.close()
201             #     IO.output_student_and_course_names(student_data=student_data)
202             # except Exception as e:
203             #     message = "Error: There was a problem with writing to the file.\n"
204             #     message += "Please check that the file is not open by another program."
205             #     IO.output_error_messages(message=message,error=e)
```

Figure 6 - converting student objects into a list of dictionaries

---

Consolidating Error Handling

Because the Data Layer, was now performing most of the validations and error handling bits, I removed these tasks from the input_student_data() method. Below is a screen shot of the updated code. (Figure 7)

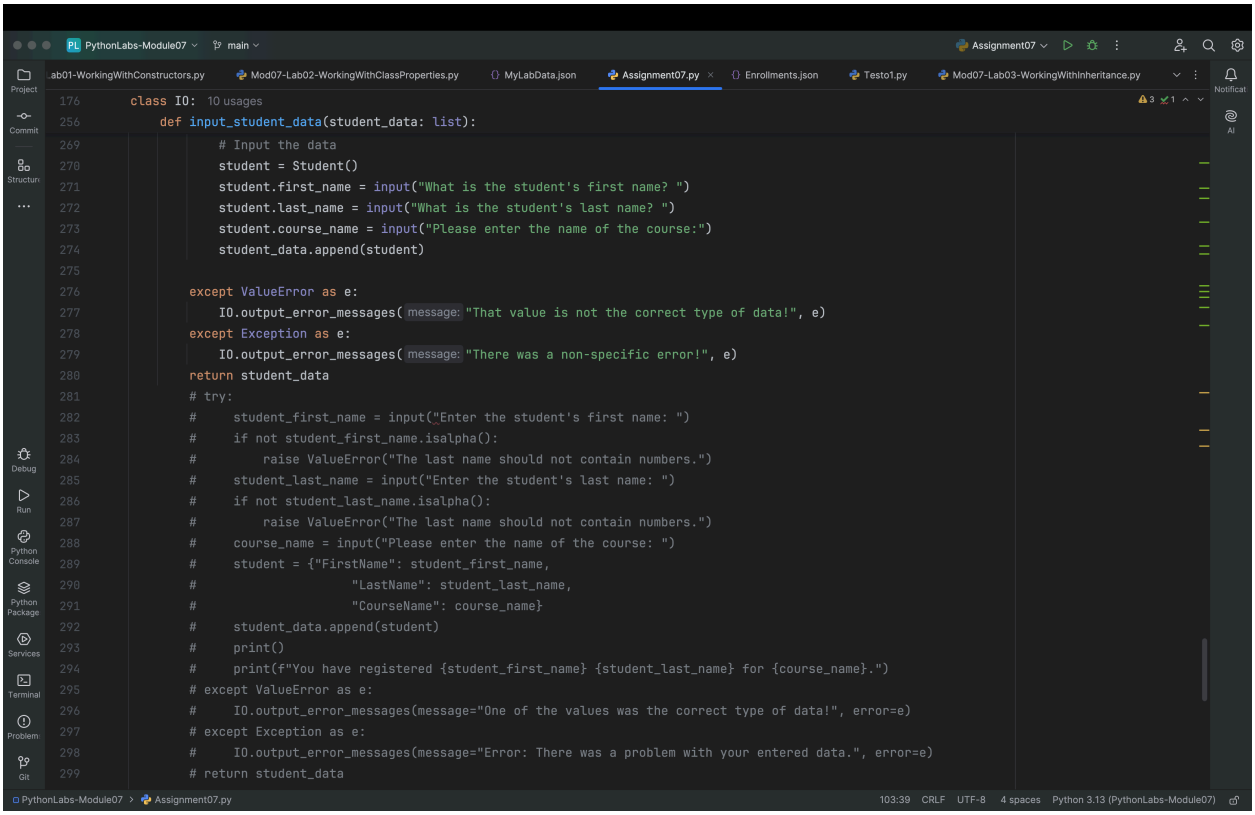

Figure 7 - Updated input_student_data() method without validation and error handling

---

Testing Program in Terminal

After successfully running the program in PyCharm, here is a scree shot of tests in Terminal. In these tests, I started by displaying current data, then selected choice one: I entered the student data in uppercase and the title() function reformatted the data.

On the second screen, I enter no data by hitting the Enter key through all the required field and the program accepts the empty data - thereby displaying a strange custom message with the user input missing. I enter the first name containing numbers and the program catches it. (Figure 8)

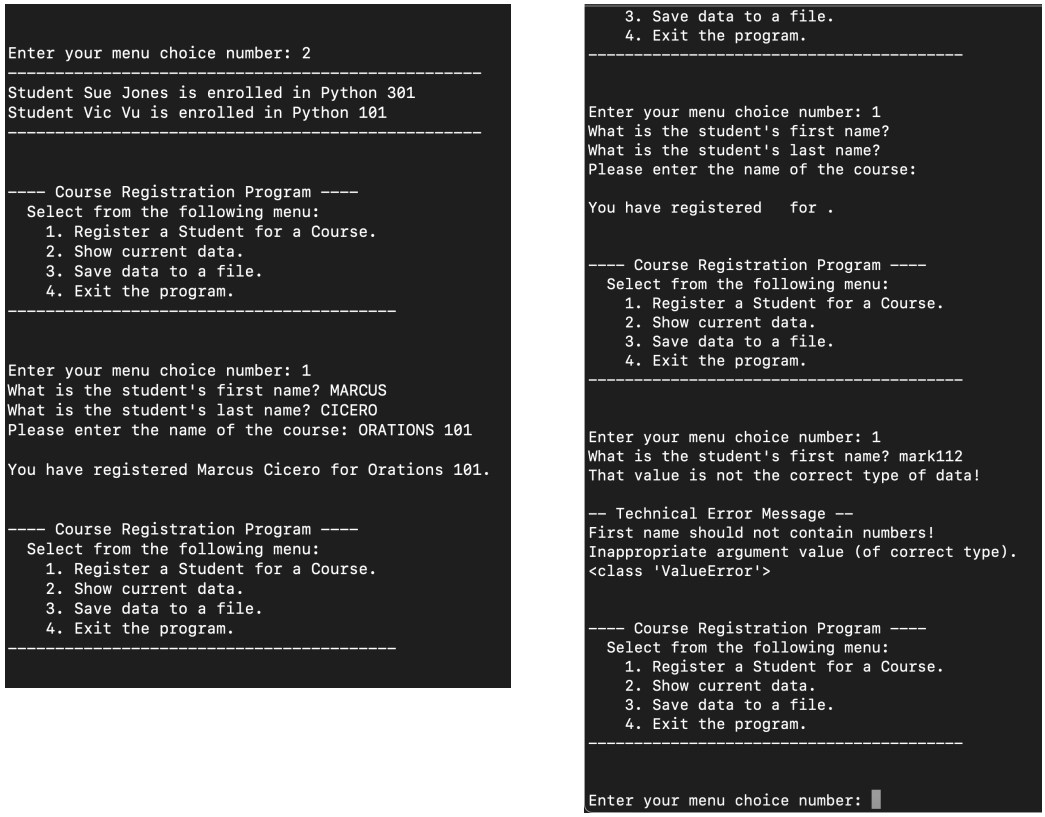

Figure 8- Testing Program in Terminal

I deleted a Curley brace in the json file data and then launched the program to test if the error handling would catch the non-json data in the file, below is the code the program displayed on launch. (Figure 9)

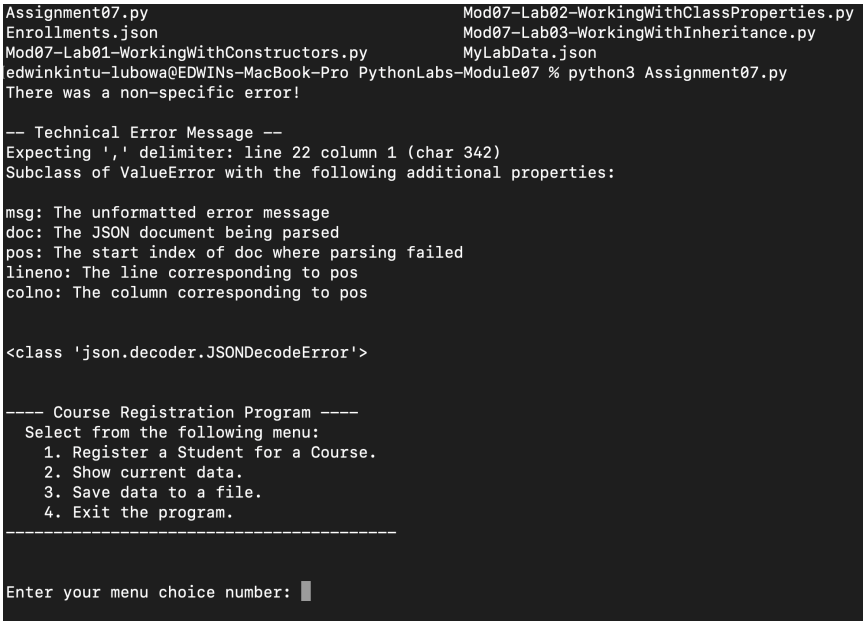

Figure 9 - Testing Program in Terminal with corrupted JSON data

Verifying data in the JSON file

Since the program was able to accept empty strings, I checked “Enrollments.json” to verify whether the initial values for the object were saved as empty strings. Below is a screen shot of the data in “Enrollments.json” showing the last student object was set with empty strings for the initial value. (Figure 10)

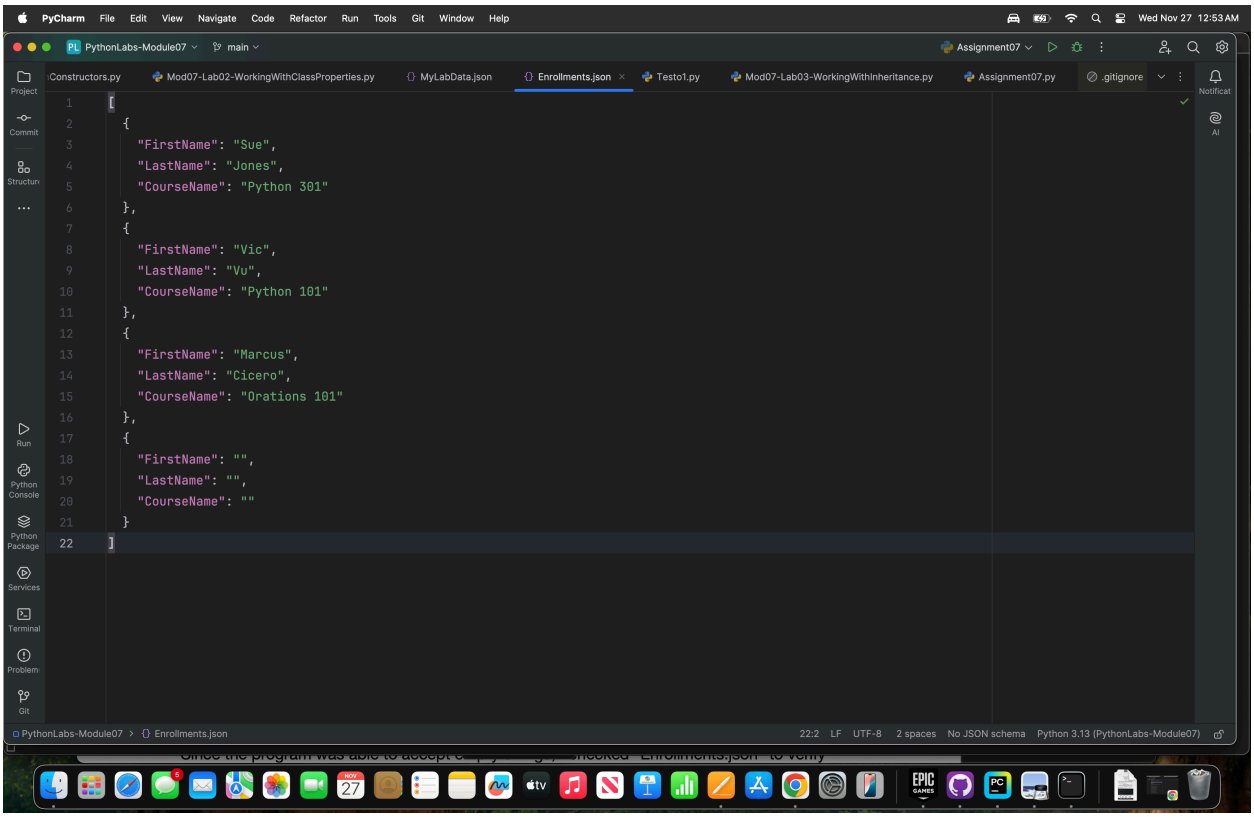

Figure 10 - results showing empty strings in “Enrollments.json”

Summary

This assignment was a confluence of many new concepts combined with some I have gotten proficient at, and others that I am still coming to grasps with. The pattern of “Separation of Concerns” played a crucial role in completing this assignment. First, there are methods in the program that I did not touch or update at all, they retained the functionality they had in the last assignment. Secondly, there are some methods that needed to be updated in order to handle the new data layer; specifically converting dictionary rows into student objects and, then,

converting student objects to dictionary lists. And finally, I had to create a new data layer, complete with supporting data processing methods.

When dealing with classes, I performed minor updates to the methods in the ‘Presentation Layer’ : I removed the validation and error handling from the input method. In the ‘Processing Layer’, I made major code updates to handle the conversions between lists of dictionaries and objects, but still retaining the overall functionality for the program. And for the ‘Data Layer’ I created the layer anew along with the supporting methods, including the concept of inherited code from the Person class to the Student class.

This was a good exercise and a practical display of how the program was made functional with varying degrees of focus on different layers.