Walden Marcus

3/12/25

IT FDN 110 A

Assignment 07

“Assignment07.py”

Introduction

“Assignment07.py” is a program designed to receive user input in regard to student registration, including the student’s first name, last name, and course name. The program can receive the information of multiple students per session of use. “Assignment07.py” reads data from a JSON file containing student information, and writes new information to it through use of the program. “Assignment07.py” utilizes classes, methods, and simple data-type transformation to handle user input.

Packages, Global Constants, Global Variables

“Assignment07.py” requires use of the *json* package, which is imported upon initiation of the program. The two global constants, *MENU* and *FILE_NAME*, respectively print the menu of options to the user and designate the file “Enrollments.json”. The two global constants, *students* and *menu_choice*, respectively hold a list of compiled student data as comma-separated strings, and the user’s menu input.

```
import json

MENU: str = '''

---- Course Registration Program ----

  Select from the following menu:

    1. Register a Student for a Course.

    2. Show current data.

    3. Save data to a file.

    4. Exit the program.

----------------------------------------

'''

FILE_NAME: str = "Enrollments.json"


students: list = []

menu_choice: str
```

Fig. 1 Package imports, global constants, and global variables

The *Person* Class

The *Person* class in "Assignment07.py" defines and handles errors pertaining to student first name and last name. The class is initiated so that each instance of *self.first_name* and *self.last_name* is named *first_name* and *last_name*. Because these variables are controlled by the constructor *__init__*, they are designated as private attributes in their "getter" methods (*first_name(self)* and *last_name(self)*). Error handling on both *first_name* and *last_name* ensure that the names only contain letters.

```
class Person:
    def __init__(self, first_name : str = "", last_name : str = ""):
        self.first_name = first_name
        self.last_name = last_name
    @property
    def first_name(self):
        return self.__first_name
    @first_name.setter
    def first_name(self, value = str):
        if value.isalpha() or value == "":
            self.__first_name = value
        else:
            raise ValueError("The first name may only contain letters.")
    @property
    def last_name(self):
        return self.__last_name
    @last_name.setter
    def last_name(self, value = str):
        if value.isalpha() or value == "":
            self.__last_name = value
        else:
            raise ValueError("The last name may only contain letters")
    def __str__(self):
        return f'{self.first_name},{self.last_name}'
```

Fig 2. The Person class, its constructor *__init__*, and the getters and setters for *first_name* and *last_name*

The *Student* Class

The *Student* Class in "Assignment07.py" is a child class of *Person*, and as such inherits the constructor *__init__* as pertains to the variables *first_name* and *last_name*. A new variable associated with *Student*,

course_name, is introduced in the constructor. Unlike *first_name* and *last_name*, *course_name* does not need to consist of only letters. As such, its setter does not raise a ValueError.

Both *Person* and *Student* override *__str__* for the purpose of returning the value of the properties *first_name* and *last_name* instead of the address of the object.

```
class Student(Person):
    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
    def course_name(self):
        return self.__course_name

    @course_name.setter
    def course_name(self, value = str):
        self.__course_name = value

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

Fig 3. The Student class, its constructor *__init__*, the getters and setters of *first_name* and *last_name*, and the override of *__str__*.

The FileProcessor Class

The *FileProcessor* class contains methods that interact with "Enrollments.JSON".

The method *FileProcessor.read_data_from_file()* reads data from a designated JSON, then loads its data to a list of dictionaries, designated *dict_list* in "Assignment07.py". Each item in *dict_list* is then converted to an iteration of *student_obj*, where *first_name*, *last_name*, and *course_name* as instances of *Student* are assigned to certain values from *dict_list* and concatenated into a comma-separated string. Each iteration of *student_obj* is appended to the list *student_data.*

The method *FileProcessor.write_data_to_file()* creates dictionary strings from the comma-separated strings by assigning each object instance of *Student* to a key in the local variable *student_json*. Each iteration of *student_json* is appended to the local variable *dict_list*. *Dict_list* is then written to "Enrollments.json" using the *json.dump* method.

```
class FileProcessor:
    @staticmethod
    def read_data_from_file(file_name : str, student_data : list):
```

```
        try:
            file = open(file_name, "r")
            dict_list = json.load(file) #returns list of dict rows
            for each in dict_list:
                student_obj : Student = Student(first_name=each["FirstName"],
                                                last_name = each["LastName"],
                                                course_name = each["CourseName"])
                student_data.append(student_obj)
            file.close()
        except FileNotFoundError as e:
            IO.output_error_messages("Data file does not exist!", e)
        except Exception as e:
            IO.output_error_messages("There was a non-specific error!", e)
        finally:
            if not file.closed:
                file.close()
        return student_data


    @staticmethod
    def write_data_to_file(file_name : str, student_data : list):
        try:
            dict_list : list = []
            for each in student_data:
                student_json : dict = {"FirstName": each.first_name,
                     "LastName": each.last_name,
                     "CourseName": each.course_name}
                dict_list.append(student_json)
            file = open(file_name, "w")
            json.dump(dict_list, file)
            file.close()
            print("The following data was saved:\n")
            IO.output_student_courses(student_data = students)
        except TypeError as e:
             IO.output_error_messages("Data not saved: Please check that the data is a valid JSON
format", e)
        except Exception as e:
            IO.output_error_messages("Data not saved: There was a non-specific error!", e)
```

```
        finally:
            if not file.closed:
                file.close()
```

Fig 4. *FileProcessor.read_data_from_file* and *FileProcessor.write_data_to_file* complete similar steps in a reverse and opposite order. Notice how these methods, like those in *IO*, are designated as static methods using the @staticmethod property. These methods also contain error handling in the case of the file not being found.

The IO Class

The *IO* class receives and manages input from the user, whether it pertains to the menu of options, or to student registration data. Additionally, *IO* displays technical error messages through the *IO.output_error_messages()* method. *IO.output_menu()* displays the menu of options to the user, and *IO.input_menu_choice()* requests and returns the choice made by the user. *IO.input_menu_choice()* will return an exception if the user enters an option that is not the numbers 1-4, which satisfies the only use case of the method in the program.

```
class IO:
    @staticmethod
    def output_error_messages(message : str, error: Exception = None):
        print(message, end="\n\n")
        if error is not None:
            print("-- Technical Error Message --")
            print(error, error.__doc__, type(error), sep = "\n")


    @staticmethod
    def output_menu(menu: str):
        print()
        print(menu)
        print()


    @staticmethod
    def input_menu_choice():
        choice = "0"
        try:
            choice = input("Enter your menu choice number: ")
            if choice not in ("1", "2", "3", "4"):
                raise Exception("Please, choose only 1, 2, 3, or 4")
        except Exception as e:
            IO.output_error_messages(e.__str__())
```

```
        return choice
```

Fig 5. *IO.output_error_messages(), IO.output_menu,* and *IO.input_menu_choice()* are all methods of the *IO* class.

IO.input_student_data collects and stores student registration information. Using *input()* statements, *IO.input_student_data()* assigns the user input to instances of *Student.* These instances of *Student* attributes are appended to a list of comma-separated strings, designated by local variable *student_data*, which is returned. Should there be a value error or a general exception, "Assignment07.py" prints custom error messages to inform the user of the error. *IO.output_student_courses* prints a message listing all the students registered thus far by calling each instance of *Student()*.

```
@staticmethod
    def input_student_data(student_data: list):
        try:
            student = Student()
            student.first_name = input("What is the student's first name? ")
            student.last_name = input("What is the student's last name? ")
            student.course_name = input("What is the student's course name? ")
            student_data.append(student)
        except ValueError as e:
            IO.output_error_messages("That value is not the correct type of data!", e)
        except Exception as e:
            IO.output_error_messages("There was a non-specific error!", e)
        return student_data


    @staticmethod
    def output_student_courses(student_data: list):
        for student in student_data:
           message = f"{student.first_name} {student.last_name} is enrolled in {student.course_name}"
            print(message)
```

Fig 6. *IO.input_student_data* and *IO.output_student_data* each utilize the a local variable called *student_data*, but the variable is contained in each method.

Program Logic

The logic of "Assignment07.py" depends on a while loop that calls certain methods based on user input. At the initiation of "Assignment07.py", "Enrollments.JSON" is read to the variable *students*. When "Assignment07" returns "1" from *IO.input_menu_choice()*, it assigns the returned value of *IO.input_student_data()* to *students*. When "Assignment07.py" returns "2", it calls

IO.output_student_courses. Upon the return of "3", *FileProcessor.write_data_to_file()* is called. Lastly, the user input of "4" to *IO.input_menu_choice()* breaks the loop and ends the program.

```
students = FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)
while True:
    IO.output_menu(menu=MENU)

    menu_choice = IO.input_menu_choice()

    if menu_choice == "1":
        students = IO.input_student_data(student_data=students)
        continue

    elif menu_choice == "2":
        IO.output_student_courses(student_data = students)
        continue

    elif menu_choice == "3":
        FileProcessor.write_data_to_file(file_name=FILE_NAME, student_data=students)
        continue

    elif menu_choice == "4":
        break
```

Fig 7. The logic of "Assignment07.py" calls various methods.

Summary

"Assignment07.py" utilizes instances of attributes, inherited code, and constructors to create a student registration program that is error-aware and concise. The simplicity of its logic increases programmer legibility, while the classes and methods each contain their complexity using local variables. Successful use of "Assignment07.py" results in the ability to enter multiple students' registration information per session, and for all registration data across all sessions to be saved on the same file.