Sabrina Fechtner

24 November 2023

IT FDN 110 B Au 23

Assignment 07

https://github.com/sfechtner42/IntroToProg-Python-Mod07

Interactive Course Registration Part 5

Introduction

This week, I was tasked to modify last week's program with more of a focus on object-oriented programming. Here, I used initializers, instance parameters, special methods, properties, and polymorphisms in each separation of concern.

Constant Variables

The constant variables have not been modified from previous submissions (Figure 1).

```
import json
from typing import TextIO, List

# Define the Data Constants
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.
-----------------------------------------
'''
# Define the Data Constants
FILE_NAME: str = "Enrollments.json"
```

Figure 1: Defining the constant variables

Data Variables

These variables are unchanged from last week (Figure 2).

```
# Define the Data Variables
students: list[Student] = []
menu_choice: str
```

Figure 2: Data variables

Object Class 1: Person

As discussed in lecture, there are classes that can extend to other classes. In this case, persons will have a first and last name regardless of their student status. Therefore, I created a Person class in order to

capture this common information. I began with the creating instance variables under "__init__" and protected those variables by adding an underscore before the variable name. Because of their protected status, I created a publicly accessible variable using @property where the first letter in each stored variable is capitalized. At this point, I built in validation of user input using setters/getters where only non-alphanumeric values will be stored (Figure 3).

```
class Person:
    def __init__(self, first_name: str, last_name: str) -> None:
        self._first_name = first_name
        self._last_name = last_name

    9 usages (2 dynamic)
    @property
    def first_name(self) -> str:
        return self._first_name.capitalize()

    7 usages (2 dynamic)
    @first_name.setter
    def first_name(self, value):
        if value.isalpha():
            self._first_name = value
        else:
            raise ValueError("The first name cannot be alphanumeric. Please re-enter the first name.")

    9 usages (2 dynamic)
    @property
    def last_name(self) -> str:
        return self._last_name.capitalize()

    7 usages (2 dynamic)
    @last_name.setter
    def last_name(self, value):
        if value.isalpha():
            self._last_name = value
        else:
            raise ValueError("The last name cannot be alphanumeric. Please re-enter the last name.")

    def __str__(self) -> str:
        return f"{self.first_name} {self.last_name}"
```

Figure 3: Person Class

Object Class 2: Student

Next, I defined the student class which type of person. Like person, students are also defined by first and last name. Therefore, the student class calls in the person class as shown by the super-init method. To make this a student class, I added the course name variable and validated that with its own setters and getters for validation. In this case, there aren't any restrictions on course name (Figure 4).

```
class Student(Person):
    def __init__(self, student_first_name: str, student_last_name: str, course_name: str) -> None:
        super().__init__(first_name=student_first_name, last_name=student_last_name)
        self.course_name = course_name

    5 usages
    @property
    def student_course_name(self) -> str:
        return self.course_name.capitalize()

    1 usage
    @student_course_name.setter
    def student_course_name(self, value):
        self.course_name = str(value)

    def __str__(self) -> str:
        return f"{super().__str__()} has been registered for {self.student_course_name}"
```

Figure 4: Student Class

Function Class 1: File Processing

The file function class still has the same operations from last week where the program would: 1) open/read and append (Figure 5) or 2) create and write to (Figure 6). This week, these static methods tracked a list of student objects instead of dictionaries defined as student. In doing so, these functions were modified so that the json will still load a list of dictionaries within the student object using manual integration.

```
@staticmethod
def read_data_from_file(file_name: str) -> list[Student]:
    """This function reads previous JSON file with student and course data..."""
    file: TextIO = None
    json_data = []
    students: list[Student] = []
    try:
        file = open(file_name, "r")
        json_data = json.load(file)
        print("Data successfully loaded from the file.")
    except FileNotFoundError:
        print("File not found, creating it...")
        with open(file_name, "w") as file:
            json.dump( obj: json_data,  fp: file)
            print("File created successfully.")
    except json.JSONDecodeError as e:
        print(f"Invalid JSON file: {e}. Resetting it...")
        with open(file_name, "w") as file:
            json.dump( obj: json_data,  fp: file)
            print("File reset successfully.")
    except Exception as e:
        print(f"An unexpected error occurred while loading data: {e}")

    for row in json_data:
        student = Student( student_first_name: row["student_first_name"],  student_last_name: row["student_last_name"],  course_name: row["course_name"])
        students.append(student)

    return students
```

Figure 5: File Processing Read Data

```python
@staticmethod
def write_data_to_file(roster: list[Student], file_name: str) -> list[Student]:
    """This function writes student and course data to JSON file..."""
    file: TextIO = None
    try:
        json_data: list[dict[str, str, str]] = []
        for student in roster:
            json_data.append({
                "student_first_name": student.first_name,
                "student_last_name": student.last_name,
                "course_name": student.student_course_name
            }
            )
        with open(file_name, "w") as file:
            json.dump( obj: json_data,  fp: file)
            print("Data successfully written to the file.")
    except TypeError as e:
        IO.output_error_messages( message: "Please check that the data is a valid JSON format",  error: e)
    except Exception as e:
        IO.output_error_messages( message: "There was a non-specific error!",  error: e)
    return roster
```

Figure 6: File Processing Function: Write Data

Function Class 2: Input/Output (IO) Data

Next, the IO functions were pulled and modified from last week. The general error handling operator and the menu IO operators (Figure 7) were unchanged from last week. The largest changes were in the student data IO operators so that they use the student object. The output student data method itterates over each varible in the student class by assigning it to another str variable (Figure 8). The input student data method uses a similar method to Figure 5 where the student object is maniupated into a list of dictionaries that can be saved onto the opened/created json file. Because this data is part of the student class, input is also subject to validataion in Figures 3 and 4 before being saved (Figure 9).

```python
@staticmethod
def output_error_messages(message: str, error: Exception = None):
    """This function displays error messages to the user..."""
    print(message, end="\n\n")
    if error is not None:
        print(f"An unexpected error occurred: {error}")

1 usage
@staticmethod
def output_menu(menu: str):
    """This function displays the menu of choices to the user..."""
    print(menu)

1 usage
@staticmethod
def input_menu_choice():
    """This function incorporates user choice from menu..."""
    choice = "0"
    try:
        choice = input("What would you like to do?: ")
        if choice not in ("1", "2", "3", "4"):
            raise Exception("Only Enter 1, 2, 3, or 4")
    except Exception as e:
        IO.output_error_messages(e.__str__())
    return choice
```

Figure 7: IO: Error Handling and Menu IO

```python
@staticmethod
def output_student_courses(student_data: list[Student]):
    """This function shows the first name, last name, and course name from the user..."""
    print("\nThe current data is:")
    for student in student_data:
        student_first_name = student.first_name
        student_last_name = student.last_name
        student_course_name = student.course_name
        print(student_first_name, student_last_name, student_course_name)
```

Figure 8: IO Function: Student Output

```python
@staticmethod
def input_student_data(student_data: List[Student]) -> List[Student]:
    """This function incorporates user choice from the menu..."""
    while True:
        # Create an instance of Student with valid initial values
        student = Student( student_first_name: "",  student_last_name: "",  course_name: "")

        student_first_name: str = input("Please enter first name: ")
        student_last_name: str = input("Please enter last name: ")
        student_course_name: str = input("Please enter the course name: ")

        try:
            student.first_name = student_first_name
            student.last_name = student_last_name
            student.student_course_name = student_course_name

            # Create a new instance with validated properties
            student = Student( student_first_name: student.first_name,  student_last_name: student.last_name,  course_name: student.student_course_name)
            student_data.append(student)

            print(
                f"You have registered {student.first_name} {student.last_name} for {student.student_course_name}.")
            break  # if registration is successful
        except ValueError as e:
            IO.output_error_messages(f"Error registering student: {e}")

    return student_data
```

Figure 9: IO Function: Student data Input

Main Program

Now the main function calls all the defined functions (Figure 10). The only major change from last week is that now I defined the students variable as a list of the Student object.

```
# Main Program:

students: list[Student] = FileProcessor.read_data_from_file(file_name=FILE_NAME)

while True:
    IO.output_menu(menu=MENU)

    menu_choice = IO.input_menu_choice()

    if menu_choice == "1":  # get student input
        students = IO.input_student_data(student_data=students)
        continue

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

    elif menu_choice == "3":  # Save data in a file
        FileProcessor.write_data_to_file(file_name=FILE_NAME, roster=students)
        continue

    elif menu_choice == "4":  # End the program
        break  # out of the while loop
```

Figure 10: Main Program

Final Result

I ran my program successfully in PyCharm and in the terminal (Figure 11) where I registered myself for Python 100 and my cat, Cinnamon, for Python 200. I intentionally entered symbols to confirm that the error handling worked. I also entered all the data in lowercase and was happy to see the display and save file had all the inputs correctly capitalized. Both methods of running the code produced a .JSON file showing Cinnamon and I registered for our respective Python courses (Figure 12).

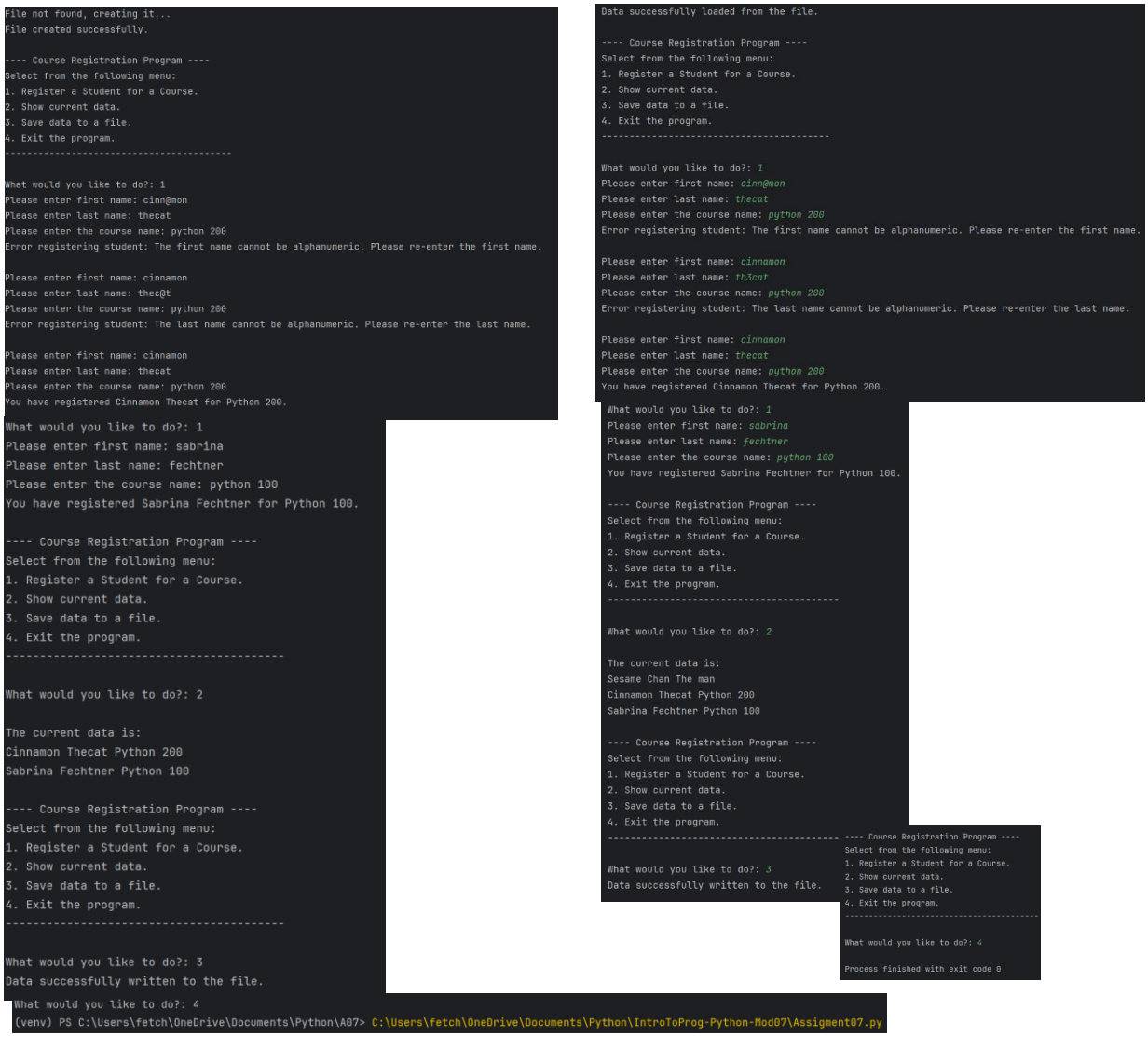

Figure 11: Terminal and Pycharm Result

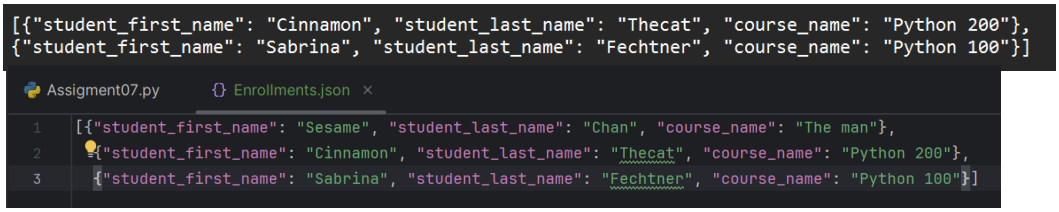

Figure 12: Result JSON files

Summary

This week, I modified my interactive course registration program from last week where I incorporated two new object classes with their own validation code. As highlighted in the lecture, separating things by concern generally results in the main program being unchanged. I came to appreciate how separating things into classes helps greatly with debugging. If I made a mistake or an error came up with a certain operation, it was much easier to look at only a section/operation rather than my entire code.