Anna-Sofiia Mykhalevych

September 10, 2024

IT FDN 110 B

Assignment 07

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

Classes and Objects

Introduction

In this assignment I learnt how to create and use classes to manage data. I focused on learning three key programming techniques to enhance my Python script: using functions classes, and adhering to the separation of concerns programming pattern. These methods played a crucial role in enhancing the structure, maintainability, and efficiency of my course registration system.

1. Defining data constants and variables

The MENU constant outlines the interactive options available for the user, which include registering a student, showing current data, saving data to a file, and exiting the program. The students variable is initialized as an empty list to hold the registered student details, and menu_choiceis prepared to capture user input. This setup enhances the organization and flexibility of the registration system, allowing for more sophisticated data handling and retrieval.

```
# 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.
-----------------------------------------
'''
FILE_NAME: str = "Enrollments_07.json"

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

Figure 1. *Defining data constants and variables*

2. Creating Person class

The code defines a Person class in Python to represent a person's data, specifically their first and last names. This class includes properties for first_name and last_name that ensure the names are formatted correctly (in title case) and contain only alphabetic characters. When a new Person object is created, the initialization method (__init__) sets the first and last names provided as arguments. The properties use getter and setter methods to control access and modifications: the setter methods check that the names consist only of letters or are empty; otherwise, they raise a ValueError. Additionally, the class includes a __str__ method to provide a nicely formatted string representation of the person's full name. This approach ensures that the name data is both well-validated and properly formatted.

```python
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:
        - RRoot, 1.1.2030: Created the class.
    """
    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.title()

    @first_name.setter
    def first_name(self, value: str):
        if value.isalpha() or value == "":
            self.__first_name = value
        else:
            raise ValueError("The last name should not contain numbers.")

    @property
    def last_name(self):
        return self.__last_name.title()

    @last_name.setter
    def last_name(self, value: str):
        if value.isalpha() or value == "":  # is character or empty string
            self.__last_name = value
        else:
            raise ValueError("The last name should not contain numbers.")

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

Figure 2. *Creating Person class*

3. Creating Student class

In this code, I created a Student class that inherits from a Person class to represent student data, specifically focusing on their first and last names. Initially, I defined the Student class with properties for first_name and last_name to manage and validate these attributes. However, I realized that both Student and other potential classes might share this common functionality related to handling names.

To make the code more reusable and maintainable, I introduced the Person class as a parent class to handle the common attributes of first_name and last_name. I then moved the initialization and the name handling logic, including the formatting and validation, into the Person class. This allowed the Student class to inherit these methods, simplifying the Student class and promoting code reuse. Now, the Student class uses the super() function to call the parent class's initializer, ensuring that the name data is set up correctly according to the rules defined in the Person class.

```
class Student(Person):
    """
    A class representing student data.

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


    ChangeLog: (Who, When, What)
    RRoot,1.1.2030,Created Class
    RRoot,1.3.2030,Added properties and private attributes
    RRoot,1.3.2030,Moved first_name and last_name into a parent class
    """
class Person:

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

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

Figure 3. *Creating Student class*

4. Class FileProcessor

n this code, I created a FileProcessor class that handles functions related to processing JSON files. The primary function I implemented is read_data_from_file, which reads data from a specified

JSON file and loads it into a list of dictionaries. I used the @staticmethod decorator because this function does not rely on any instance-specific data; it just reads from a file.

Inside the function, I employed a try block to open the file and load its contents using Python's json module. If the file is not found, a FileNotFoundError is caught, and a custom error message is generated using IO.output_error_messages. I also added a more general exception handler to catch any other issues that may arise while reading the file, providing a more informative error message.

```
class FileProcessor:
    """
    A collection of processing layer functions that work with Json files

    ChangeLog: (Who, When, What)
    RRoot,1.1.2030,Created Class
    """
    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """ This function reads data from a json file and loads it into a list of dictionary rows

        ChangeLog: (Who, When, What)
        RRoot,1.1.2030,Created function

        :return: list
        """

        try:
            with open(file_name, "r") as file:
                student_data = json.load(file)

        except FileNotFoundError:
            IO.output_error_messages(message="Error: File not found.", error=f"File {file_name} was not found.")

        except Exception as e:
            IO.output_error_messages(message="Error: There was a problem with reading the file.", error=e)


        return student_data
```

Figure 4. *Class FileProcessor*

5. Write data to file method (FileProcessor class)

In this code, I added a write_data_to_file method to the FileProcessor class. This method writes data to a JSON file from a list of dictionaries, which is passed as the student_data parameter. I marked it as a @staticmethod since it does not depend on instance-specific data.

To handle file writing, I used a try block to open the specified file in write mode ("w") and then used json.dump to serialize the student_data into JSON format. After successfully writing the data, I call IO.output_student_and_course_names to output relevant information. I also added an except block to catch any exceptions that may occur during the file writing process, providing an error

message if something goes wrong. Additionally, I used a finally block to ensure that the file is properly closed, even if an error occurs, to avoid any potential resource leaks.

```
@staticmethod
def write_data_to_file(file_name: str, student_data: list):
    """ This function writes data to a json file with data from a list of dictionary rows

    ChangeLog: (Who, When, What)
    RRoot,1.1.2030,Created function


    :return: None
    """

    try:
        file = open(file_name, "w")
        json.dump(student_data, file)
        file.close()
        IO.output_student_and_course_names(student_data=student_data)
    except Exception as e:
        message = "Error: There was a problem with writing to the file.\n"
        message += "Please check that the file is not open by another program."
        IO.output_error_messages(message=message,error=e)
    finally:
        if file.closed == False:
            file.close()
```

***Figure 5**. Write data to file method (FileProcessor class)*

6. Class IO

In this segment of my Python project, I expanded the functionality of the IO class by adding a method output_error_messages. This method is crucial for providing clear and informative feedback to users when errors occur. It displays a custom message, and if an exception is provided, it further elaborates on the technical details of the error, including the exception's documentation and type.

```
                file.close()
class IO:
    """
    A collection of presentation layer functions that manage user input and output

    ChangeLog: (Who, When, What)
    RRoot,1.1.2030,Created Class
    RRoot,1.2.2030,Added menu output and input functions
    RRoot,1.3.2030,Added a function to display the data
    RRoot,1.4.2030,Added a function to display custom error messages
    """

    @staticmethod
    def output_error_messages(message: str, error: Exception = None):
        """ This function displays the a custom error messages to the user

        ChangeLog: (Who, When, What)
        RRoot,1.3.2030,Created function

        :return: None
        """
        print(message, end="\n\n")
        if error is not None:
            print("-- Technical Error Message -- ")
            print(error, error.__doc__, type(error), sep='\n')
```

Figure 6. *IO class*

7. Enhancing IO class

In this part of my project, I enhanced the IO class by adding methods for displaying the menu and handling menu choice input from users, emphasizing user interaction within the application.

1. **Outputting the Menu:** The output_menu static method neatly presents the menu options to the user. I added extra print statements to insert blank lines before and after the menu, improving the visual layout and making it easier for users to read and make a selection.
2. **Handling User Input:** The input_menu_choice method prompts the user to enter their menu choice, ensuring that it is one of the valid options ("1", "2", "3", "4"). If an invalid option is entered, an exception is raised. I used error handling to catch this exception and employed the previously created output_error_messages method to provide a clear error message without exposing unnecessary technical details to the user.

```
    @staticmethod
    def output_menu(menu: str):
        """ This function displays the menu of choices to the user

        ChangeLog: (Who, When, What)
        RRoot,1.1.2030,Created function


        :return: None
        """
        print()  # Adding extra space to make it look nicer.
        print(menu)
        print()  # Adding extra space to make it look nicer.

    @staticmethod
    def input_menu_choice():
        """ This function gets a menu choice from the user

        ChangeLog: (Who, When, What)
        RRoot,1.1.2030,Created function

        :return: string with the users choice
        """
        choice = "0"
        try:
            choice = input("Enter your menu choice number: ")
            if choice not in ("1","2","3","4"):  # Note these are strings
                raise Exception("Please, choose only 1, 2, 3, or 4")
        except Exception as e:
            IO.output_error_messages(e.__str__())  # Not passing e to avoid the technical message

        return choice
```

Figure 7. *Enhancing IO class*

8. Output_student_and_course_namesstatic method

In this part of the project, I further developed the IO class by implementing the output_student_and_course_namesstatic method. This method is tailored to display student and course information clearly to the user.

Here's how it works:

- **Formatting and Presentation:** The method starts and ends with a line of dashes to visually separate the student data output from other text in the console. This helps in keeping the user interface clean and easy to read.
- **Handling Empty Data:** Before attempting to display any student information, the method checks if the student_data list is empty. If it is, a message "No student data to display." is printed, informing the user that there are no records to show.
- **Data Output:** For each student in the student_data list, the method prints the student's first and last names along with the course name they are enrolled in, formatted in a clear and readable manner.

```python
@staticmethod
def output_student_and_course_names(student_data: list):
    """ This function displays the student and course names to the user

    ChangeLog: (Who, When, What)
    RRoot,1.1.2030,Created function

    :return: None
    """

    print("-" * 50)
    if not student_data:  # Checks if the list is empty or None
        print("No student data to display.")
        return
    for student in student_data:
        print(f'Student {student["FirstName"]} '
              f'{student["LastName"]} is enrolled in {student["CourseName"]}')
    print("-" * 50)
```

Figure 8. *Output_student_and_course_namesstatic method*

9. Collect student details from the user

In this section of the project, I designed the input_student_data static method within the IO class to interactively collect student details from the user. The method gathers the student's first name, last name, and the course they wish to enroll in, incorporating several checks to ensure the data integrity:

- **Data Validation:** The method validates that the first and last names contain only alphabetical characters by using the isalpha() method. If the validation fails, it raises a ValueError with a message indicating that the names should not contain numbers or special characters.
- **Error Handling:** The method is wrapped in a try-except block to handle any exceptions that might arise during data input. This includes handling specific ValueErrors raised by validation failures and more general exceptions that could occur during the input process.
- **Feedback to User:** Upon successfully registering a student, it confirms the registration by printing a message. In case of an error, it utilizes the output_error_messages function from the same class to display appropriate error messages to the user.

```python
@staticmethod
def input_student_data(student_data: list):
    """ This function gets the student's first name and last name, with a course name from the user

    ChangeLog: (Who, When, What)
    RRoot,1.1.2030,Created function

    :return: list
    """

    try:
        student_first_name = input("Enter the student's first name: ")
        if not student_first_name.isalpha():
            raise ValueError("The last name should not contain numbers.")
        student_last_name = input("Enter the student's last name: ")
        if not student_last_name.isalpha():
            raise ValueError("The last name should not contain numbers.")
        course_name = input("Please enter the name of the course: ")
        student = {"FirstName": student_first_name,
                   "LastName": student_last_name,
                   "CourseName": course_name}
        student_data.append(student)
        print()
        print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
    except ValueError as e:
        IO.output_error_messages(message="One of the values was the correct type of data!", error=e)
    except Exception as e:
        IO.output_error_messages(message="Error: There was a problem with your entered data.", error=e)
    return student_data
```

Figure 9. *Collect student details from the user*

10. Main execution loop

In this segment of my Python script, I implemented the main execution loop for the course registration system, integrating various functionalities into a cohesive workflow. The script begins by loading existing student data from a JSON file. It then presents a menu to the user, captures their choice, and performs actions based on their selection, such as registering a new student, displaying current data, saving updates to a file, or exiting the program. Each menu option triggers a specific function that handles data collection, display, or storage, ensuring the application is interactive and user-friendly.

```
        return student_data
# Extract the data from the file
students = FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)

# Present and Process the data
while (True):

    # Present the menu of choices
    IO.output_menu(menu=MENU)

    menu_choice = IO.input_menu_choice()

    # Input user data
    if menu_choice == "1":  # This will not work if it is an integer!
        students = IO.input_student_data(student_data=students)
        continue

    # Present the current data
    elif menu_choice == "2":
        IO.output_student_and_course_names(students)
        continue

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

    # Stop the loop
    elif menu_choice == "4":
```

Figure 10. *Main execution loop*

Summary

In this assignment, I built a course registration program using Python's object-oriented programming features. I started by creating a Person class to handle basic details like first and last names, which helped me understand how to use properties to manage and validate data. I then extended this to a Student class to represent students more specifically, allowing me to reuse and build upon the existing Person class structure.