Phuong Tran (Eric)

November 29, 2024

IT FDN 110 A

Assignment 07

GitHubURL: erictran03/erictran03-IntroToProg-Python-Mod07

# Classes and Objects

## Introduction

Assignment 07 builds upon the foundational work of Assignment 06, introducing object-oriented programming (OOP) principles such as class inheritance, property validation, and improved program structure through modular design. These enhancements addressed the limitations of the previous assignment, such as redundant validation and less structured data handling, resulting in a more maintainable and scalable program.

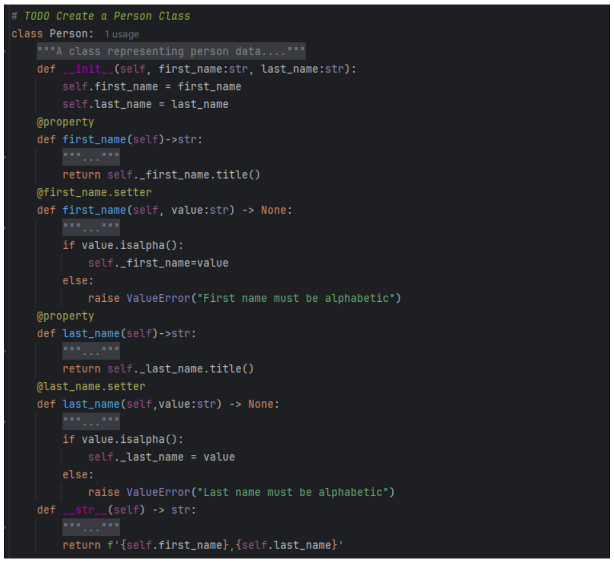
## How OOP Changed the Implementation

Although the program's execution steps—registering students, displaying data, saving data, and exiting—remained unchanged, the implementation saw major improvements due to the adoption of OOP.

### 1. Object-Oriented Design

In Assignment 07, two primary classes were introduced:

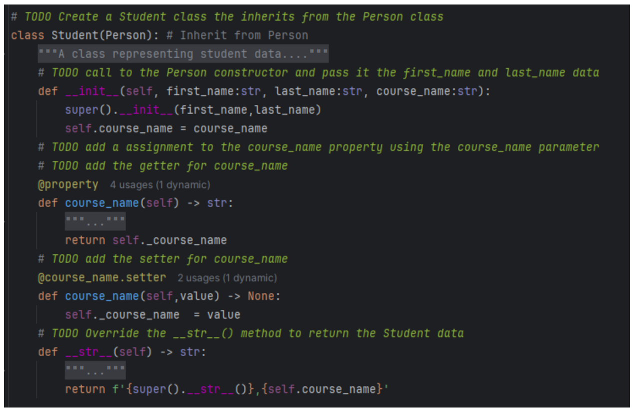
- **Person Class** (Figure 1)**:** Handles first_name and last_name attributes, including validation through properties.



```
# TODO Create a Person Class
class Person:  1 usage
    """A class representing person data...."""
    def __init__(self, first_name:str, last_name:str):
        self.first_name = first_name
        self.last_name = last_name
    @property
    def first_name(self)->str:
        """..."""
        return self._first_name.title()
    @first_name.setter
    def first_name(self, value:str) -> None:
        """..."""
        if value.isalpha():
            self._first_name=value
        else:
            raise ValueError("First name must be alphabetic")
    @property
    def last_name(self)->str:
        """..."""
        return self._last_name.title()
    @last_name.setter
    def last_name(self,value:str) -> None:
        """..."""
        if value.isalpha():
            self._last_name = value
        else:
            raise ValueError("Last name must be alphabetic")
    def __str__(self) -> str:
        """..."""
        return f'{self.first_name},{self.last_name}'
```

*Figure 1: Person Class*

- **Student Class** (Figure 2)**:** Inherits from Person and adds a course_name attribute.



```python
# TODO Create a Student class the inherits from the Person class
class Student(Person): # Inherit from Person
    """A class representing student data...."""
    # TODO call to the Person constructor and pass it the first_name and last_name data
    def __init__(self, first_name:str, last_name:str, course_name:str):
        super().__init__(first_name,last_name)
        self.course_name = course_name
    # TODO add a assignment to the course_name property using the course_name parameter
    # TODO add the getter for course_name
    @property  4 usages (1 dynamic)
    def course_name(self) -> str:
        """..."""
        return self._course_name
    # TODO add the setter for course_name
    @course_name.setter  2 usages (1 dynamic)
    def course_name(self,value) -> None:
        self._course_name  = value
    # TODO Override the __str__() method to return the Student data
    def __str__(self) -> str:
        """..."""
        return f'{super().__str__()},{self.course_name}'
```

*Figure 2: Student Class*

This design allowed for data encapsulation and reuse, ensuring consistent behavior across the program.

By using classes, I replaced repetitive dictionary handling with structured objects. This approach centralized logic, reduced redundancy, and made the program easier to maintain.

## 2. Centralized Validation

In Assignment 06, input validation was performed manually in multiple places, leading to scattered and repetitive code. In Assignment 07, validation was integrated directly into the classes using properties, as shown in Figure 1 and Figure 2.

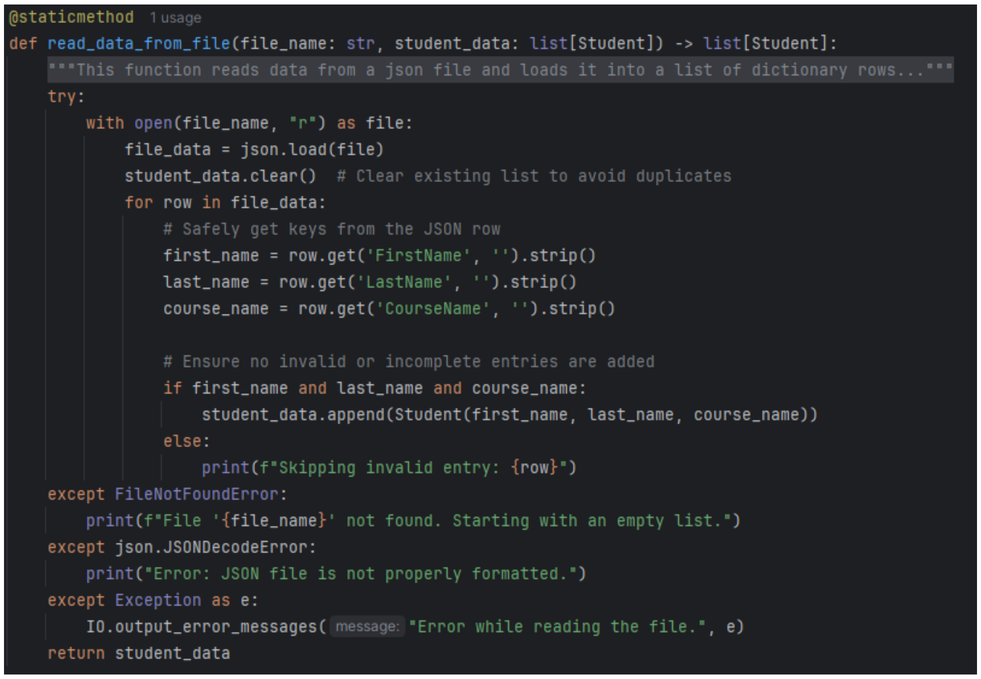
Properties ensured consistent validation at the point of object creation, making the program more robust. Invalid data could not enter the system, reducing the need for error-checking throughout the code.

3. Improved File Handling

In Assignment 07, read_data_from_file() and write_data_to_file() were updated to handle Student objects instead of raw dictionaries, ensuring validated and structured data throughout the program.

- **read_data_from_file()** (Figure 3): Converts dictionaries from the JSON file into Student objects, clearing existing data to avoid duplication. Gracefully handles missing files by starting with an empty list.
- **write_data_to_file()** (Figure 4): Converts Student objects into dictionaries before saving to JSON, ensuring compatibility with the file format. Includes error handling and provides success messages.

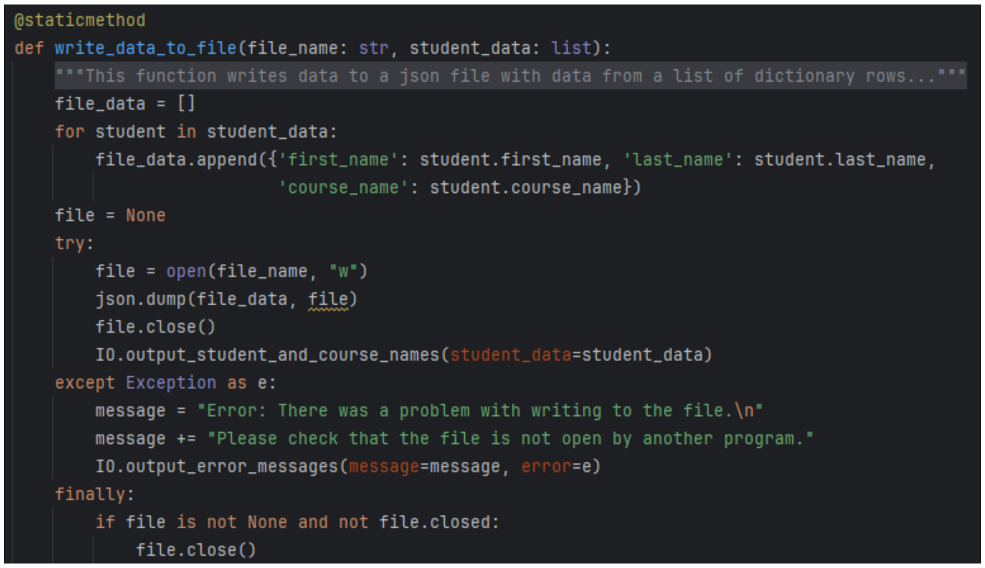
These updates align file handling with OOP principles, ensuring validated data, robust error handling, and seamless object conversion.



```
@staticmethod 1 usage
def read_data_from_file(file_name: str, student_data: list[Student]) -> list[Student]:
    """This function reads data from a json file and loads it into a list of dictionary rows..."""
    try:
        with open(file_name, "r") as file:
            file_data = json.load(file)
            student_data.clear()  # Clear existing list to avoid duplicates
            for row in file_data:
                # Safely get keys from the JSON row
                first_name = row.get('FirstName', '').strip()
                last_name = row.get('LastName', '').strip()
                course_name = row.get('CourseName', '').strip()

                # Ensure no invalid or incomplete entries are added
                if first_name and last_name and course_name:
                    student_data.append(Student(first_name, last_name, course_name))
                else:
                    print(f"Skipping invalid entry: {row}")
    except FileNotFoundError:
        print(f"File '{file_name}' not found. Starting with an empty list.")
    except json.JSONDecodeError:
        print("Error: JSON file is not properly formatted.")
    except Exception as e:
        IO.output_error_messages( message: "Error while reading the file.", e)
    return student_data
```

*Figure 3: Load JSON data as dictionaries*



```
@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..."""
    file_data = []
    for student in student_data:
        file_data.append({'first_name': student.first_name, 'last_name': student.last_name,
                          'course_name': student.course_name})
    file = None
    try:
        file = open(file_name, "w")
        json.dump(file_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 is not None and not file.closed:
            file.close()
```

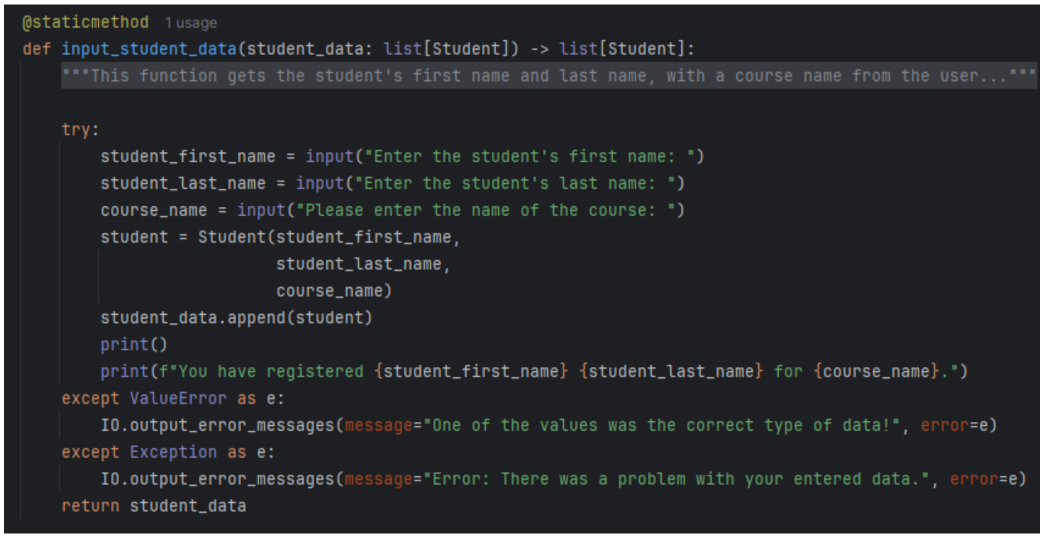
*Figure 4: Write data to a JSON file*

## 4. Modular Program Structure

The program was divided into three main components:

- **Person** and **Student**: Handle data encapsulation and validation.
- **FileProcessor**: Manages file operations, including reading and writing JSON data.
- **IO**: Handles user interaction, such as displaying menus and error messages

The **input_student_data()** method was updated to handle exceptions and integrate seamlessly with the validation logic in the Student class. User input is collected, and any errors raised by invalid data (e.g., non-alphabetic names) are caught and displayed to the user (Figure 5).


```
@staticmethod  1 usage
def input_student_data(student_data: list[Student]) -> list[Student]:
    """This function gets the student's first name and last name, with a course name from the user..."""

    try:
        student_first_name = input("Enter the student's first name: ")
        student_last_name = input("Enter the student's last name: ")
        course_name = input("Please enter the name of the course: ")
        student = Student(student_first_name,
                          student_last_name,
                          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 5: Input student data method*

The updated IO class improves error handling, provides clear feedback to the user, and works in harmony with the Student class for centralized validation. This makes the program easier to maintain and more user-friendly.

## Testing the Program

I tested the program by running it in both PyCharm and Terminal and entering multiple student records. I checked for the following scenarios:

- Registering students with valid and invalid input.
- Displaying the list of registered students, including cases where no students are registered.
- Saving the data to the JSON file and verifying its contents in a text editor.
- Handling file errors, such as missing files or corrupted JSON data.

All tests were successful, and the program handled errors gracefully, providing helpful feedback to the user.

## Summary

Assignment 07 retained the functionality of Assignment 06 but transformed the implementation using object-oriented programming. By introducing classes, properties, and modular design, the program became more robust, scalable, and maintainable. These changes addressed the limitations of the previous version, ensuring data validation, separation of concerns, and better file handling. This assignment demonstrated the practical benefits of OOP and prepared me for building more complex Python applications.