Name: Student-Py-Q
Date: 03/08/25
Course: IT FDN 110 A Wi 25: Foundations of Programming Python
Assignment: 07

GitHub URL:
https://github.com/Student-PyQ/IntroToProg-Python-Mod07

Classes and Objects

Introduction

Assignment 7 is all about building the foundation for understanding object oriented programming (OOP) concepts and best practices. In the last assignment we covered functions, classes, and separations of concerns. But for this assignment the focus is on using data classes to demonstrate data encapsulation, isolation, reusability, and abstraction as benefits to OOP. As well as, understanding how inheritance works in objects. I approached this assignment in three steps:

- Step 1 - Understanding Data Classes
- Step 2 - Working with Data in the Abstract
- Step 3 - Overriding Methods in Objects

Step 1 - Understanding Data Classes

For the first step of this assignment, I used the starter code file and created an empty Person and Student class (**See Figure 1.1**). These two classes will be the data classes used for designing constructor methods. Constructor methods are like functions that automatically initialize base object attributes and the class variable values when the class is called. Since the Person and Student classes will serve as data classes, I added attributes, constructors, getters (properties), and setters. I learned that getters are not actually decorated with the getter syntax in Python, but rather the property decorator is used to annotate a method that makes the class data attributes accessible. The setter decorator is used to allow the class attributes to mutate or change its value inside the method (**See Figure 1.2**).

Figure 1.1 - Empty Person and Student Classes

```python
# TODO Create a Person Class
class Person:
    """
    A class representing a person data item.
    Properties:
        first_name: str
        last_name: str
    Changelog:
        Student-PyQ, 3/9/2025, Created class.
    """
    pass

# TODO Add first_name and last_name properties to the constructor (Done)
# TODO Create a getter and setter for the first_name property (Done)
# TODO Create a getter and setter for the last_name property (Done)
# TODO Override the __str__() method to return Person data (Done)
class Student:
    """
    A class representing student data item.
    Properties:
        first_name: str
        last_name: str
        course_name: str
    Changelog:
        Student-PyQ, 3/9/2025, Created class.
```

Figure 1.2 - Getters and Setters

```python
class Person:
    """
    # TODO Add first_name and last_name properties to the constructor (Done)
    def __init__(self, first_name: str = '', last_name: str = ''):
        self.first_name = first_name
        self.last_name = last_name

    # TODO Create a getter and setter for the first_name property (Done)
    @property  # (Use this decorator for the getter or accessor)  1 usage
    def first_name(self):
        return self.__first_name.title()

    @first_name.setter  1 usage
    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.")

    # TODO Create a getter and setter for the last_name property (Done)
    @property  1 usage
    def last_name(self):
        return self.__last_name.title()  # formatting code

    @last_name.setter  1 usage
```

Step 2 - Working with Data in the Abstract

For the second step, I learned about abstraction and encapsulation. Both programming concepts support object oriented programming. Abstraction simplifies what attributes define an object and encapsulation focuses on how that abstraction is performed. A benefit of encapsulation is preventing unintended changes or coding errors by restricting direct access to class attributes. An example of this is adding string formatting or error handling logic statements within the setter methods so that logic is encapsulated within the data attribute method (**Figures 2.1**). I can reference a student object data attribute with built in error checking without having to write additional code when setting data values from input statements.

Figure 2.1 - Error Handling for Person Data Attribute

```python
@last_name.setter  1 usage
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.")
```

```python
input_student_data  Cc W .*  2/2
class IO:  10 usages
    def input_student_data(student_data: list):

        try:
            #Modify to use object instead...
            student = Student()
            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.")"""
            student.course_name = input("Please enter the name of the course: ")
```

Step 3 - Overriding Methods in Objects

The last step is putting the magic to work with inheritance core concepts and overriding methods in objects to simplify code. For example, in the main code of the program I modified the *read_data_from_file* method of the FileProcessor class to return a list of Student objects instead of a list of dictionary collection types. The benefit of doing this is less code to manage because the Student data class inherited most of the Person data class attributes, specifically first_name and last_name (**See Figure 3.1**). These data attributes are derived from the super or parent class, Person, by using the __int__() magic methods in the Student data class. Any special coding, error handling is encapsulated with the Person class attributes where it belongs closest to the data fields. By doing so any data value setting error checking is conducted once at the parent class only.

Another example of object oriented programming is the capability of overriding methods. In this case with the Student data class, the inclusion of course name data attribute is an example of overriding the Person data class attributes (**See Figure 3.2**). This can be done explicitly adding new data attributes through the use of additional getters and setters. Or through the use of __str__() magic method.

Figure 3.1

```python
# TODO Create a Student class the inherits from the Person class (Done)
class Student (Person):  3 usages
    """
    A class representing student data item.
    Properties:
        first_name: str
        last_name: str
        course_name: str
    Changelog:
        Student-PyQ, 3/9/2025, Created class.
    """
    # TODO call to the Person constructor and pass it the first_name and last_name data (Done)
    def __init__(self, first_name: str = '', last_name: str = '', course_name: str = ''):
        super().__init__(first_name=first_name, last_name=last_name)
        # TODO add a assignment to the course_name property using the course_name parameter (Done)
        self.course_name = course_name
```

Figure 3.2

```python
class Student (Person):  3 usages
    def course_name(self):
        return self.__course_name
    # TODO add the setter for course_name (Done)
    @course_name.setter  4 usages (2 dynamic)
    def course_name(self, value: str):
        if value.startswith("Python") or value == "":
            self.__course_name = value
        else:
            raise ValueError("Must be a Python course.")
    # TODO Override the __str__() method to return the Student data (Done)
    def __str__(self):
        return f'{self.first_name},{self.last_name},{self.course_name}'
```

Summary

In summary, the use of classes and objects in programming languages can be quite powerful and provide more reusable code in the long run. It may not appear like it at first because it requires thoughtful design and planning to organize code for mangability. Object oriented programming is a best practice that has stood the test of time.

Console Screen & PyCharm Prints

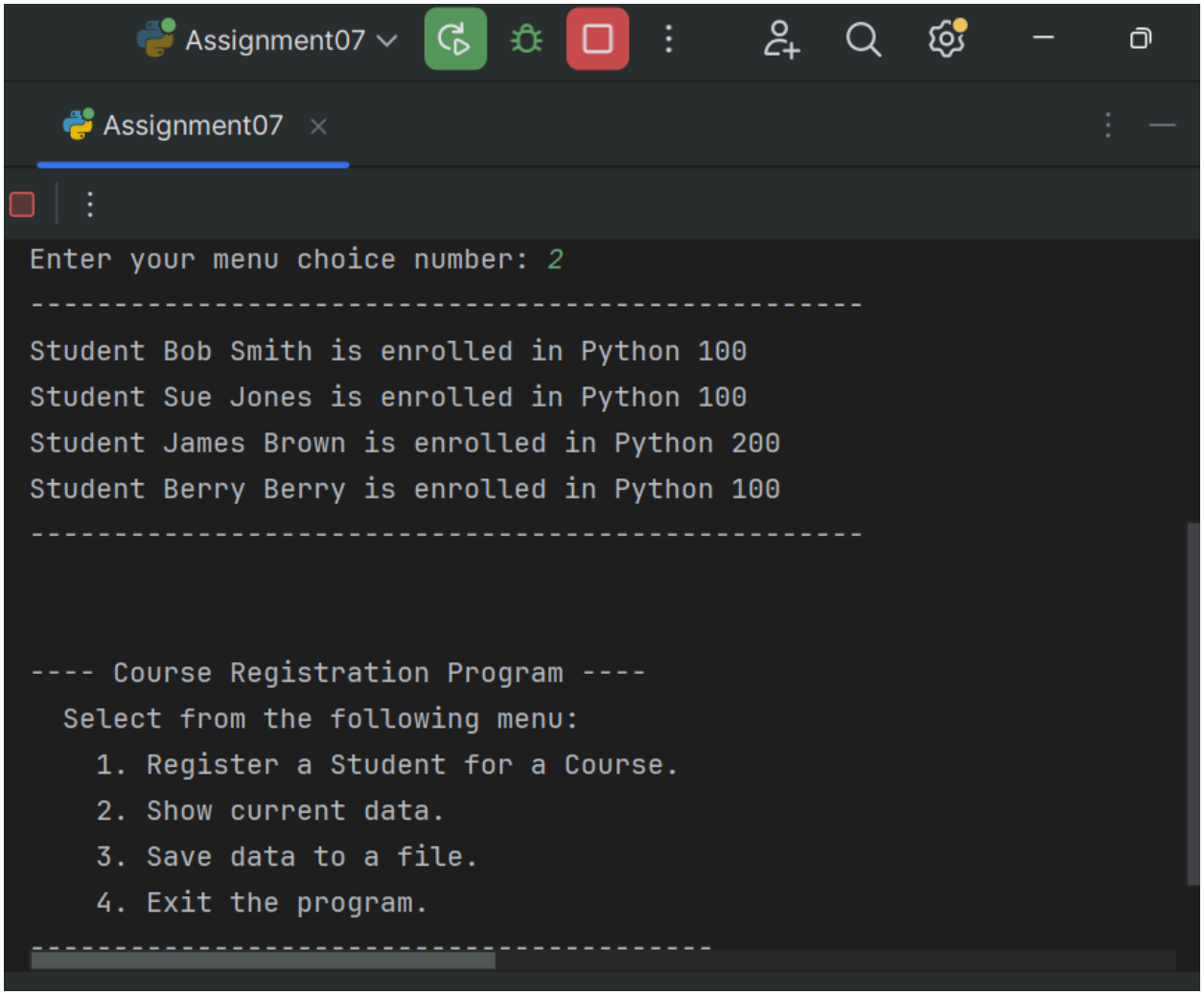
```
Command Prompt - python /

    2. Show current data.
    3. Save data to a file.
    4. Exit the program.
----------------------------------------


Enter your menu choice number: 2
--------------------------------------------------
Student Bob Smith is enrolled in Python 100
Student Sue Jones is enrolled in Python 100
Student James Brown is enrolled in Python 200
Student Berry Berry is enrolled in Python 100
--------------------------------------------------


---- 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.
----------------------------------------


Enter your menu choice number:
```



References

Root, R. (2025). Module 07- Programming Basics. In IT FDN 110 A Winter 2025, *Introduction to Programming with Python*, (pp. 1-15) University of Washington