

Sarah West

11/25/24

IT FDN 110 A

Assignment 07

<https://github.com/sarahmwest/IntroToProg-Python-Mod07>

Classes and Objects

Introduction

In this assignment, I updated the script from Assignment 06 to include objects and classes that make use of inheritance code from parent classes. Overall, in this assignment I further utilized Separations of Concern (SoC) to continue to improve readability, usability, and maintainability of the program by making use of the concepts of encapsulation and abstraction. I used constructors, attributes, and properties to achieve this goal.

Creating the Script

The Data Layer

I began by opening the “Assignment07-starter.py” that was provided us. Some basic pseudo-code and starter code was already written that resembled the final code in Assignment 06. I updated the script header to contain my information.

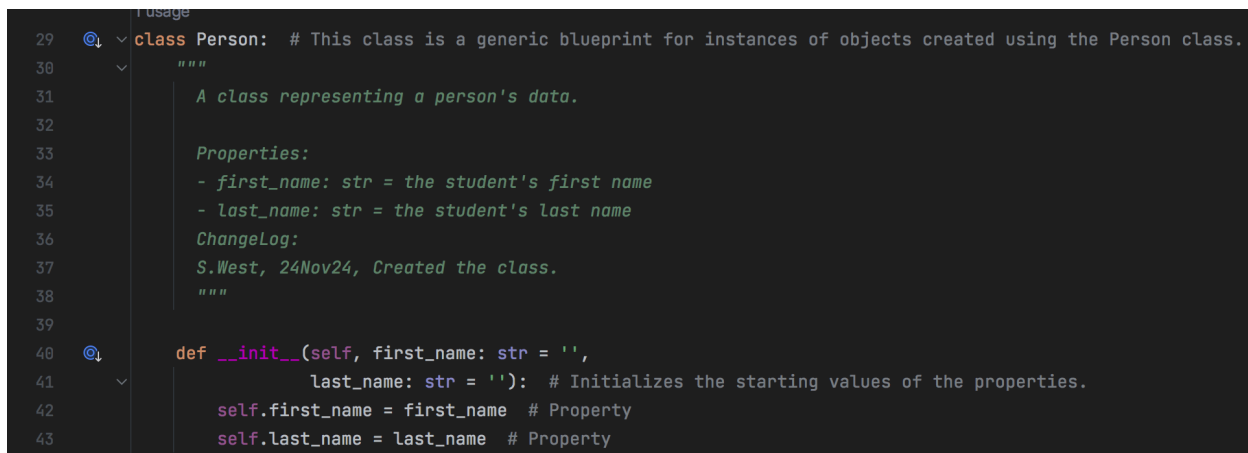
In this script, I further utilized the Separation of Concerns (SoC) pattern to organize the code to improve readability, maintainability, and scalability of the code. I included objects in the Data Layer which simplifies the Processing and Presentation Layer by creating two new data classes: Person and Student (**Figure 1**).

```
11      # -----DATA-----#
12      # Constants
13      > MENU: str = '''...'''
22      FILE_NAME: str = "Enrollments.json"
23
24      # Global Variables
25      students: list = [] # A table of student data.
26      menu_choice: str # Hold the choice made by the user.
27
28
29      1 usage
30      > class Person:...
97
98
99      > class Student(Person):...
```

Figure 1. A screenshot showing the collapsed data classes in the data layer: Person and Student.

Person Class

To begin, I created a Person class. This class acts as a blueprint to create new object instances of a person, which eliminates the need to repeat code and logic over and over in the script. Instead, I can call the class by using dot (.) notation to create objects of class “Person”. **Figure 2** shows the docstring of the class and the constructor for the class.



```
29  class Person: # This class is a generic blueprint for instances of objects created using the Person class.
30      """
31          A class representing a person's data.
32
33          Properties:
34          - first_name: str = the student's first name
35          - last_name: str = the student's last name
36          ChangeLog:
37          S.West, 24Nov24, Created the class.
38      """
39
40      def __init__(self, first_name: str = '',
41                  last_name: str = ''): # Initializes the starting values of the properties.
42          self.first_name = first_name # Property
43          self.last_name = last_name # Property
```

Figure 2. A screenshot showing the Person class docstrings and constructor method.

Note that throughout the script, I updated the docstrings with new information summarizing the code I added to each method and class. Next, I added a constructor method (function), also known as the initializer, which is automatically called when an object of the class is created. The purpose of the constructor is to initialize the attribute values to default values. The initializer uses the syntax: `def __init__(self, attribute1, attribute2,...)`, where the `def` specifies the definition of the method, the `__init__` is the keyword for the constructor method, `self` is the name of the first instance of the attributes in the initializer that attaches the attributes defined in the constructor to the instances of the objects (`self` should always be the first parameter defined in the constructor), and then the attributes (variables that hold data specific to an object, also known as “fields”). In this case, the attributes which will apply to all instances of the objects created from the person class are: “`first_name`” and “`last_name`”.

After defining the constructor, the properties need to be defined so we can call them when we create instances of person objects. I used “`self.first_name = first_name`” and “`self.last_name = last_name`” statements to accomplish this. `Self` refers to this instance of these properties in the constructor. The values `first_name` and `last_name` are passed to the attributes so when the first and last names are defined by the user later on when the user creates an object of the Person class, they are “encapsulated” and unique within the object. Encapsulation is a type of SoC that focuses on how “abstraction” is done, and abstraction focuses on what an object does and restricts direct access to the object’s components which enhances data integrity. In other words, encapsulation hides the internal implementation of code, such as validation, from external code that uses the class, and abstraction is the act of hiding the code by binding the data (attributes and properties) and methods that construct the class.

Figure 3 shows the “getter” and “setter” properties that manage the attribute data. The “getter” or “accessor” is responsible for getting the data, and the “setter” or “mutator” is responsible for setting the data. Getter and setter property methods also allow you to perform data validation and error handling as well as setting the privacy of attributes.

```
48     @property # Getter
49     def first_name(self):
50         """
51         Returns the first name as a title.
52         :return : first name with proper formatting (string).
53         """
54         return self.__first_name.title() # Returns the inputted name in title case.
55
56     # The double underscore above means that the attribute is private and cannot be changed by outside code.
57
58     9 usages (8 dynamic)
59     @first_name.setter
60     def first_name(self, value: str):
61         """
62         Sets the first name with validation.
63         :param value: (string)
64         :return : None
65         """
66         # The value parameter is what will be entered later on by the user.
67         if value.isalpha() or value == '': # Validation code for if the entered value is not a string/alphanumeric.
68             self.__first_name = value
69         else:
70             raise ValueError("The name should only contain letters.")
```

Figure 3. A screenshot showing the getter and setter properties in the person class.

The getter property is defined by the statement “@property” and the setter with the statement “@attribute_name.setter” with the functions defined in the next line. The parameter self is included in both the getter and setter function to specify this instance of the object. Note that the getter and setter function names must match for them to work as a pair. The getter and setter methods are where the privacy of the attributes are defined using the double underscore “__” before the name of the attribute. This means that the attribute cannot be changed by outside code and helps with data integrity. The getter method in **Figure 3** has formatting code that returns the inputted first_name in title case to help it look nicer. The setter method, on the other hand, has the data validation. Specifically, if the first name entered is not alphanumeric or a string value, a ValueError will be raised to let the user know that the name type is wrong. The value parameter in the setter method will be the string entered by the user when the input() method is called. The person class also has getter and setter methods for the last_name attribute, and have the same code and logic as for the first name, with modifications to reflect the last name.

```
91  @.
92  def __str__(self): # Overrides the default string method that prints the memory address to add a custom comment!
93      """
94      The string function for Person.
95      :return : The string as a csv value.
96      """
97      return f'{self.__first_name}, {self.__last_name}'
```

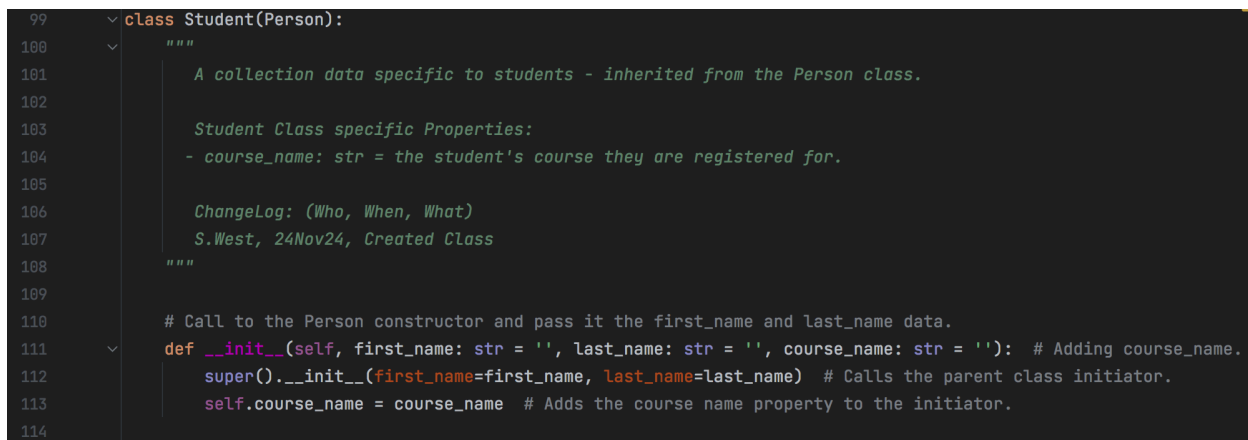
Figure 4. A screenshot showing the custom string method in the person class.

Lastly, I added a custom string method (**Figure 4**) in the Person class that overrides one of the “Magic Methods” in Python’s built-in “Object” class that is automatically called in response to

specific operations. The `__str__()` method in Python is responsible for returning human-readable strings of objects. The default `__str__()` method returns the object's location in memory, which is not very useful for most humans running the program. My custom string method will override the default one and return a string with the Person's first and last name as csv when invoked.

Student Class

Now that I have the Person Class in place that manages all the variables/parameters/attributes related to a generic person, I added the Student class for student specific information (**Figure 5**). In this case, the extra information I want included is the course name that the student is registered for.



```
99  class Student(Person):
100      """
101          A collection data specific to students - inherited from the Person class.
102
103          Student Class specific Properties:
104          - course_name: str = the student's course they are registered for.
105
106          ChangeLog: (Who, When, What)
107          S.West, 24Nov24, Created Class
108      """
109
110      # Call to the Person constructor and pass it the first_name and last_name data.
111      def __init__(self, first_name: str = '', last_name: str = '', course_name: str = ''): # Adding course_name.
112          super().__init__(first_name=first_name, last_name=last_name) # Calls the parent class initiator.
113          self.course_name = course_name # Adds the course name property to the initiator.
114
```

Figure 5. A screenshot of the Student class with inheritance from the person class.

The Student class makes use of “inherited” code from the parent class (the super class, Person). Inheritance is a concept in object-oriented programming (OOP) languages such as Python where a new class (the child class, the sub-class) can inherit data and behaviors and methods from an existing class (the parent class) to eliminate the need of adding the same code to the new class. Inheritance drastically simplifies the code in the script and makes it more readable and user friendly. In my case, I am specifying that the Student class inherits methods from the Person class with the syntax “`class Student(Person)`”. Note that in the Person class, no inheritance was specified, meaning that it inherited code from Python’s root class “Object”. After defining the docstring (**Figure 5**), I added the constructor to the Student class with an added parameter, `course_name`. Then, I called the Person constructor with the “`super().__init__(first_name=first_name, last_name=last_name)`” statement and defined the attribute `course_name` using the same logic as the attributes in the parent class.

Similar to the parent class, I added getter and setter properties for the private attribute `course_name` using the same logic minus the validation in the setter. Lastly, I updated the `__str__()` method in the Student class to override the string method in the Person class to include the course name (see **Figure 6** below).

```

def __str__(self): # Override the default string that prints the memory address to add a custom comment!
    """
    The string function for Student.
    :return : The string as a csv value.
    """
    return f'{self.__first_name}, {self.__last_name}, {self.__course_name}'

```

Figure 6. A screenshot of the string method in the Student class.

The Processing Layer - FileProcessor Class

Now that the data layer has been updated to include the classes Person and Student, I updated the code in some of the methods in the FileProcessor Class to work with objects instead of dictionaries. The docstrings in the code have been updated to reflect the changes.

Method: read_data_from_file

Figure 7 shows the method that reads the data from the file and loads it into the parameter, student_data. Note that I collapsed the docstring in **Figure 7** to show the updated code in a single screenshot.

```

150 @staticmethod
151 def read_data_from_file(file_name: str, student_data: list):
152     """This function reads data from a json file and loads it into a list of student objects..."""
153     list_of_dictionary_data: list = [] # Stores the JSON file data as a list of dictionaries.
154     student_object: object # The new object variable that loads data into student_data.
155     try:
156         file = open(file_name, "r")
157         list_of_dictionary_data = json.load(file) # The load function returns a list of dictionary rows.
158         for student in list_of_dictionary_data:
159             student_object = Student(first_name=student["FirstName"],
160                                     last_name=student["LastName"],
161                                     course_name=student[
162                                         "CourseName"]) # Converts dictionaries to objects.
163             student_data.append(student_object) # Adds the converted data from the JSON file into the parameter.
164             file.close()
165     except FileNotFoundError as e: # I added more custom error handling.
166         IO.output_error_messages(message="Text file must exist before running this script!", e)
167     except Exception as e:
168         IO.output_error_messages(message="There was a non-specific error reading the file!", e)
169     finally:
170         if file.closed == False:
171             file.close()
172     return student_data

```

Figure 7. A screenshot showing the read_data_from_file method and it's updated code.

Since the code in the starter file is written to work with JSON files and not objects, I defined two new local variables: student_object and list_of_dictionary_data. The list_of_dictionary_data variable equals the json.load() function thereby loading the variable with the dictionary data in the file. Next, I updated the for loop to iterate though the data stored in the list_of_dictionary_data variable to create new objects that represent the student data: first name, last name, and course name. I accomplished this by setting a new variable student_object equal to a new student object that pulls the first_name, last_name, and course_name variables from the JSON dictionary data in the file by calling the class Student(); the methods in the Student class and it's parents class are being initiated to format the student objects. Lastly, the data is loaded into student_object and is appended to the student_data parameter and the file is closed and saved.

Method: write_data_to_file

Next, I updated the write_data_to_file method to work with student objects instead of dictionaries.

```
184 @staticmethod
185 def write_data_to_file(file_name: str, student_data: list):
186     """This function writes data to a json file with data from a list of student objects..."""
187
188     try:
189         list_of_dictionary_data: list = []
190         for student in student_data: # Convert list of student objects to list of dictionary rows.
191             # Formats the student's data into dictionary rows for each student in student_data.
192             student_json: dict \
193                 = {"FirstName": student.first_name, "LastName": student.last_name,
194                   "CourseName": student.course_name}
195             list_of_dictionary_data.append(student_json) # Appends data in the newly created
196             # dictionary row for each student to the list_of_dictionary_data variable.
197         file = open(file_name, "w")
198         json.dump(list_of_dictionary_data, file) # Writing the data stored as a dictionary to file.
199         file.close()
200         IO.output_student_and_course_names(
201             student_data=student_data) # Shows the user all the data that has been saved.
202     except Exception as e:
203         message = "Error: There was a problem with writing to the file.\n"
204         message += "Please check that the file is not open by another program."
205         IO.output_error_messages(message=message, error=e)
206     finally:
207         if file.closed == False:
208             file.close()
```

Figure 8. A screenshot showing the write_data_to_file method and its collapsed docstring.

In the write_data_to_file method shown in **Figure 8**, I declared the list_of_dictionary_data local variable which I used to convert the student objects back to dictionaries so the data can be written to a JSON file. I edited the for loop to use dot (.) notation to call the first_name, last_name, and course_name from the student objects stored in student_data and convert them to dictionaries; a row of dictionary data for one student is stored in the student_json variable. I then appended the dictionary data stored in student_json to the list_of_dictionary_data where data for each student is stored. After iterating through each student, the data is written to the file using the json.dump statement and the file is closed and saved. I added the output method statement to show the user the data saved to the file.

The Presentation Layer - Class IO

In the presentation layer that hold the IO class, I did not change the code in the following methods because the code already works with object data: output_error_messages, output_menu, and input_menu_choice. The remaining methods output_student_and_course_names and input_student_data required editing to work with object data.

Method: output_student_and_course_names

Figure 9 shows the method displaying the student information to the user. In this method, I updated the code in the for loop to work with student objects using dot (.) notation to call the student's first and last name and course name that are stored in the parameter `student_data`.

```
285     @staticmethod
286     > def output_student_and_course_names(student_data: list):
296         """This function displays the student and course names to the user..."""
297
298         print("-" * 50)
298         > for student in student_data: # Updated the print statement to call the student objects.
299         >     print(f'Student {student.first_name} '
300               f'{student.last_name} is enrolled in {student.course_name}')
301         print("-" * 50)
```

Figure 9. A screenshot showing the updated code in the `output_student_and_course_names` method.

Method: `input_student_data`

I updated the `input_student_data` method in **Figure 10** to work with objects by setting the student variable equal to the student objects created from class `Student()`. Then, I set the first name, last name, and course name values in the student objects equal to the user input. The inputted data is then appended to the parameter `student_data` that stores all the instances of student objects. Note that validation is written into the class properties and therefore I removed most of the error handling specific to inputted data that was present in the starter code.

```
303     @staticmethod
304     > def input_student_data(student_data: list):
305     >     """This function gets the student's first name and last name, with a course name from the user..."""
315
316     try:
317         student = Student() # The variable student creates objects from the class Student.
318         # Sets the names of the object instance from class student equal to the user input.
319         student.first_name = input("What is the student's first name? ")
320         student.last_name = input("What is the student's last name? ")
321         student.course_name = input("Please enter the course name: ")
322         student_data.append(student) # Adds the student object to the list of objects stored in student_data.
323     except ValueError as e:
324         IO.output_error_messages(message="One of the values was the correct type of data!", error=e)
325     except Exception as e:
326         IO.output_error_messages(message="Error: There was a problem with your entered data.", error=e)
327     return student_data # The newly entered data is stored in the student_data variable as a list of objects.
328
```

Figure 10. A screenshot showing the `input_menu_choice` method.

Main Body of the Script

Method Calls

The only change I made to the method calls in the main body of the script was remove the error handling in the else block; this was redundant since I added validation in the data layer as already discussed. I also added a method call to show the student data after registering a student for the course (menu choice 1).

Testing the Script in PyCharm and MacOS

I tested the script by running it in both PyCharm and my MacOS terminal and I am happy to report it runs great! I tested various error handling methods and all work as they should, and I confirmed that the data is successfully written to the JSON file.

Summary and Conclusion

In this assignment, I updated the code that was present in Assignment 06 to work with objects, inherited classes, and added a more robust data layer with constructors, attributes, and properties. I also demonstrated the use of encapsulation and abstraction by creating classes that encapsulate code and uses properties to provide validation and error handling. I am happy to report that my script runs great!