

Bryan Pannell

5/29/2024

IT FDN 110 A

Assignment 07

<https://github.com/bpannell12/IntroToProg-Python-Mod07>

Data Classes

Introduction

This week's assignment continues our work with classes by adding on data handling classes. Just as we grouped functions in classes last week to perform file processing and input/output, this week we added two additional classes, Person and Student to handle our data and even some error handling. This continued the streamlining of the main body of our script and further organized how things are aligned. The code is even more modular. In the end, the script still delivers the same results for the user but does so in a very different way. This week, we took the previous work and re-organized it using these classes and functions.

Working with JSON files

To be able to work with JSON files, the script must import Python's built-in JSON capabilities. Being able to do this greatly simplifies working with this file type and allows for a wide variety of operations to be performed using the data and files.

```
7  
8     import json  
9
```

Figure 1: Simple code to import Python's built-in JSON functionality

Script Structure and Comments

Proper documentation is even more important as more complexity is added. In addition to the comments and pseudo code previously recorded, specific entries for classes and functions are essential. These elements require adequate explanations for others to be able to understand the code and how it operates.

```

10 # Define the Data Constants
11 MENU: str = ''
12 ---- Course Registration Program ----
13     Select from the following menu:
14         1. Register a Student for a Course.
15         2. Show current data.
16         3. Save data to a file.
17         4. Exit the program.
18     -----
19     '''
20 FILE_NAME: str = "Enrollments.json"
21
22 # TODO Create a Person Class
23 1 usage
24 @ class Person:
25     """
26     A collection of functions that create the Person records using the first and last names.
27     Includes error handling for invalid name entries.
28     ChangeLog: (Who, When, What)
29     BPannell, 5.27.2024, Created Class
30     """
31     # TODO Add first_name and last_name properties to the constructor
32     @ def __init__(self, first_name: str, last_name: str):
33         self.first_name = first_name
34         self.last_name = last_name
35
36     # TODO Create a getter and setter for the first_name property
37     2 usages
38     @property
39     def first_name(self) -> str:
40         """ Returns the first name with a starting capital letter
41         :return: First name, ensuring first letter is capitalized
42         """
43         return self._first_name.title()

```

Figure 2: Example of pseudo code and explanatory notes

Defining Constants and Variables

Because the two new data classes are added into the mix this week, there was a little shifting of the definitions for the variables this week. Constants are still declared upfront as seen in Figure 3 below. The other global variables are moved down just before the main body. This is done to position them appropriately for use.

```

10      # Define the Data Constants
11      MENU: str = '''
12      ---- Course Registration Program ----
13      Select from the following menu:
14          1. Register a Student for a Course.
15          2. Show current data.
16          3. Save data to a file.
17          4. Exit the program.
18      -----
19      '''
20      FILE_NAME: str = "Enrollments.json"
21

```

```

290
291      # Define the Data Variables
292      students: list[Student] = [] # a table of student data
293      menu_choice: str # Hold the choice made by the user.
294
295      # Start of main body
296
297      # When the program starts, read the file data into a list of lists (table)
298      # Extract the data from the file
299      students = FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)
300

```

Figure 3: Defining constants and variables

Defining the Classes and Functions

In addition to the classes that were established last week, this week two data classes were added into the mix. Last week, we used the FileProcessor and IO classes to bundle functions to perform the operations needed by the script. This week we added two data handling classes: Person and Student. The Person class sets up the person's record by handling the work around getting the first and last name of the registrants. The Student class inherits the first and last names from the Person class and adds on the course name. These classes combine to comprise the registration record that is stored in the data table in the file. The Person class even handles the error handling work of checking the first and last name to see if they are in the proper format. The figure below shows the code to set the two new functions up:

```

22      # TODO Create a Person Class
23      1 usage
24      class Person:
25          """
26          A collection of functions that create the Person records using the first and last names.
27          Includes error handling for invalid name entries.
28          Changelog: (Who, When, What)
29          Bpamell, 5.27.2024, Created Class
30          """
31          # TODO Add first_name and last_name properties to the constructor
32          def __init__(self, first_name: str, last_name: str):
33              self.first_name = first_name
34              self.last_name = last_name
35
36          # TODO Create a getter and setter for the first_name property
37          2 usages
38          @property
39          def first_name(self) -> str:
40              """ Returns the first name with a starting capital letter
41              :return: First name, ensuring first letter is capitalized
42              """
43              return self._first_name.title()
44
45          1 usage
46          @first_name.setter
47          def first_name(self, value: str) -> None:
48              """ Sets the first name and checks for value errors
49              :param value: The first name to set
50              """
51              if value.isalpha():
52                  self._first_name = value
53              else:
54                  raise ValueError("First name contains only alphabet letters.")
55
56

```

```

57      class Person:
58
59          # TODO Create a getter and setter for the last_name property
60          2 usages
61          @property
62          def last_name(self) -> str:
63              """ Returns the last name with a starting capital letter
64              :return: Last name, ensuring first letter is capitalized
65              """
66              return self._last_name.title()
67
68          1 usage
69          @last_name.setter
70          def last_name(self, value: str) -> None:
71              """
72              Sets the last name and checks for value errors
73              :param value: The last name to set
74              """
75              if value.isalpha():
76                  self._last_name = value
77              else:
78                  raise ValueError("Last name contains only alphabet letters.")
79
80          # TODO Override the __str__() method to return Person data
81          def __str__(self) -> str:
82              return f'{self.first_name},{self.last_name}'
83
84

```

```

77 class Student(Person):
78     """
79     A collection of functions that combine the Person records with the course name to create a complete register
80     record.
81
82     ChangeLog: (Who, When, What)
83     B.Pannell, 5.27.2024, Created Class
84     """
85
86
87     def __init__(self, first_name: str, last_name: str, course_name: str):
88         super().__init__(first_name, last_name) # TODO call to the Person constructor and pass it the first name
89         self.course_name = course_name
90
91     # TODO add an assignment to the course_name property using the course_name parameter
92     # TODO add the getter for course_name
93     4 usages
94     @property
95     def course_name(self) -> str:
96         """
97         Returns the course name
98         :return: Course name
99         """
100         return self.__course_name
101
102     # TODO add the setter for course_name
103     1 usage
104     @course_name.setter
105     def course_name(self, value) -> None:
106         self.__course_name = value
107
108     # TODO Override the __str__() method to return the Student data
109     def __str__(self) -> str:
110         return f'{super().__str__()}, {self.course_name}'

```

Figure 4: Example of code establishing data classes

Because new data classes were added this week, we must make sure that any other functions in the FileProcessor and IO classes that reference the previous version's dictionaries are edited to instead point to the new data set generated by the data classes. The following FileProcessor functions required editing for this: read_data_from_file and write_data_to_file. In the IO class, these functions required changes to perform their operations correctly: input_student_data and output_student_and_course_names. Below is an example of the edited version of the input function:

```

184 class IO:
185
186     1 usage
187
188     @staticmethod
189     def input_student_data(student_data: list[Student]) -> list[Student]:
190         """ This function collects user input: first name, last name and course name.
191
192         ChangeLog: (Who, When, What)
193         B.Pannell, 5.27.2024, Created function
194
195         :param student_data: list of rows to be filled with input data
196
197         :return: list
198         """
199
200         try:
201             student_first_name = input("Enter the student's first name: ")
202             student_last_name = input("Enter the student's last name: ")
203             course_name = input("Please enter the name of the course: ")
204             student = Student(student_first_name,
205                               student_last_name,
206                               course_name)
207             student_data.append(student)
208             print()
209             print(f'You have registered {student_first_name} {student_last_name} for {course_name}.')
210         except ValueError as e:
211             IO.output_error_messages(message="One of the values was the correct type of data!", error=e)
212         except Exception as e:
213             IO.output_error_messages(message="Error: There was a problem with your entered data.", error=e)
214         return student_data

```

Figure 5: Example of edited code to utilize new data set

The Main Body

Once all the classes and functions are set up, the main body of the script is needed to create the linkage to make everything work. The main body calls the functions in each class at the appropriate point to perform each of their assigned tasks.

The first thing that needs to be done is to read and extract the data from a specified file:

```
295     # Start of main body
296
297     # When the program starts, read the file data into a list of lists (table)
298     # Extract the data from the file
299     students = FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)
```

Figure 6: Example code to call the function to read and extract data from a file

Just as in previous assignments, we use a while loop to be able to account for multiple entries:

```
301     # Present and Process the data
302     while (True):
303
```

Figure 7: Python code example to start a “while” loop

While the While loop continues to execute, the menu system needs to be functional. To accomplish this, the functions related to displaying and collecting the user’s menu choice are called on each loop iteration:

```
304         # Present the menu of choices
305         IO.output_menu(menu=MENU)
306
307         menu_choice = IO.input_menu_choice()
```

Figure 8: Python code to call the menu call the menu display function and get user selection

```

----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 selection for the options above:

```

Figure 9: Menu display

Depending on the menu option selected by the user, the “if” and “elif” statement for that option triggers the function call for that specific task. The user can enter a menu option on each iteration of the loop. Invalid entries are handled by error handling in the specific functions. The figure below shows this structure:

```

308
309     # Input user data
310     if menu_choice == "1": # This will not work if it is an integer!
311         students = IO.input_student_data(student_data=students)
312         continue
313
314     # Present the current data
315     elif menu_choice == "2":
316         IO.output_student_and_course_names(students)
317         continue
318
319     # Save the data to a file
320     elif menu_choice == "3":
321         FileProcessor.write_data_to_file(file_name=FILE_NAME, student_data=students)
322         continue
323
324     # Stop the loop
325     elif menu_choice == "4":
326         break # out of the loop
327
328     print("Program Ended")

```

Figure 10: Python code example for menu processing using function calls

Running the script produces the same results as last week’s assignment, but the code is more efficiently structured. Virtually all the file processing, data handling, error handling, and input/output operations are handled in the classes and functions. The main body exists solely to call upon those classes and functions to make things work. Below are samples of the output:

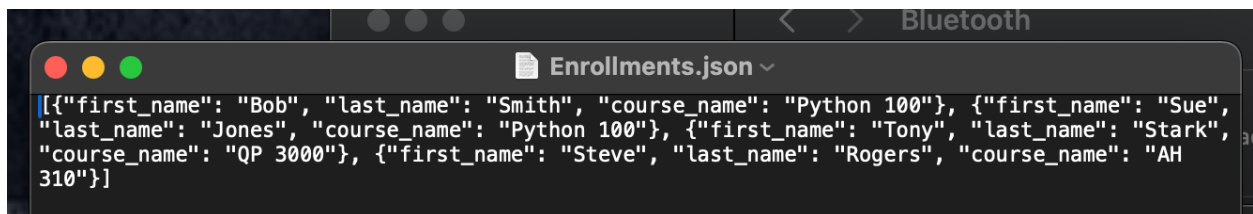
```

---- 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.
-----

What would you like to do: 3
The following data was saved to file!
Student Bob Smith is enrolled in Python 100
Student Sue Jones is enrolled in Python 100
Student Steve Rogers is enrolled in Ah 310
Student Tony Stark is enrolled in QP 3000

```

Figure 11: Script report for saving data to a JSON file



```

[{"first_name": "Bob", "last_name": "Smith", "course_name": "Python 100"}, {"first_name": "Sue",
"last_name": "Jones", "course_name": "Python 100"}, {"first_name": "Tony", "last_name": "Stark",
"course_name": "QP 3000"}, {"first_name": "Steve", "last_name": "Rogers", "course_name": "AH
310"}]

```

Figure 12: JSON data file sample

Once the loop is exited, adding a simple `print()` line with a message such as “Program Ended” lets the user know the session is over.

```

---- 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.
-----

What would you like to do: 4
Program Ended

```

Figure 13: Sample exit message

Making sure that everything runs correctly

Error handling is built into several of the functions to handle issues that might arise. The following functions have error handling:

- read_data_from_file()
- write_data_to_file()
- input_menu_choice()
- input_student_data()

This error handling identifies issues that occur with input, output, or file handling. For error handling, the above functions call the `IO.output_error_messages()` function. Compared with last week's script, you may notice that error handling for ensuring that the first and last names contain only alphabet letters was removed from `input_student_data()`. This error handling was moved to the Person data class. Below is an example of error handling within one of the data class functions, specifically the setter for the `first_name`:

```
42  
1 usage  
43 @first_name.setter  
44 def first_name(self, value:str) -> None:  
45     '''Sets the first name and checks for value errors  
46     :param value: The first name to set  
47     '''  
48     if value.isalpha():  
49         self._first_name = value  
50     else:  
51         raise ValueError("First name contains only alphabet letters.")  
52
```

Figure 14: Sample of structured error handling built into setter for a data class function

Summary

This week's assignment took the existing script and added two new data handling classes. By adding these two new classes, the code becomes even more modular. The main body of the script becomes much more focused on calling the functions within the classes and letting them do the heavy lifting. These classes are organized by what they do. This makes the code more easily followed in how it performs. The key is to get the various parts correctly linked together and talking to each other; passing on data and tasks so that they work together to generate the final results. These results are the same, but the route to this destination has changed.