Patrick Moynihan
May 11, 2024
Course: Foundations of Programming: Python
Assignment: 05
GitHub link: https://github.com/Pyronious/IntroToProg-Python-Mod05

Exercise: Advanced Collections and Error Handling

Contents

Introduction

The goal of this exercise is to learn about working with dictionaries, JSON files, and error handling. We'll read data from a JSON file into a list of dictionaries, allow the user to display and append the data, and write the list data back out to a JSON file. We'll also add data validation and error handling such as missing files.

Data Preparation

Our first step is to define the constants and variables we will be using throughout the script.

Constants

We start by declaring our constants. We have some familiar strings from last week, plus a new list as shown in line 22 of Figure 1.

```
12  MENU: str = '''
13  ------ Course Registration Program ------
14    Select from the following menu:
15      1. Register a Student for a Course.
16      2. Show current data.
17      3. Save data to a file.
18      4. Exit the program.
19  ----------------------------------------
20  '''
21  FILE_NAME: str = "Enrollments.json"
22  KEYS: list = ["FirstName", "LastName", "CourseName"]
```

Figure 1: Declaring constants.

We'll use the **KEYS** list to validate the keys we find inside the JSON file when we read it.

Variables

Most of our variable declarations are the same as last week. The one key difference is that **student_data** is dictionary now instead of a list. We'll be storing the data as a list of dictionaries instead of a list of lists like we did last week.

```
24  # Define the Data Variables
25
26  student_first_name: str = ''  # Holds the first name of a student entered by the user.
27  student_last_name: str = ''  # Holds the last name of a student entered by the user.
28  course_name: str = ''  # Holds the name of a course entered by the user.
29  json_data: str = ''  # Holds combined string data separated by a comma.
30  file: IO  # Holds a reference to an opened file.
31  menu_choice: str = ''  # Hold the choice made by the user.
32  student_data: dict = {}  # Dictionary of data for a single student
33  students: list = []  # List of data for all students
34  saved: bool = True # Tracks whether newly added data has been saved
```

Figure 2: Declaring variables.

On line 32 we declare a dictionary variable named **student_data**, which will hold the student's first name, last name, course name using key-value pairs.

On line 33 we declare a list named **students**, which will hold a list of **student_data** dictionaries. This "2-dimensional" list effectively represents a table (or spreadsheet, like Excel) and is a handy way to store records that can be described in a table-like format.

It might be helpful to visualize each element in **student_data** as a column in a single row of the table. Each value has a "key" which identifies which column the value belongs to. Likewise we can

think of the elements in the **students** list as the rows in the table. Figure 3 shows how we can visualize the data structure we're looking to create.

	A	B	C	D
1		FirstName	LastName	CourseName
2	students[0]	Frodo	Baggins	Adventure Management
3	students[1]	Samwise	Gamgee	Culinary Arts
4	students[2]	Meriadoc	Brandybuck	Hobbit Shenanigans

Figure 3: Visualizing our 2-dimensional list of dictionaries.

So, if we wanted to find out the last name of the 2nd student in the list, we'd go looking inside **students[1]** because it holds the **student_data** dictionary for the 2nd student in the list. Then we could look inside that dictionary using the key **LastName** to retrieve the student's last name from the dictionary.

Loading data from a JSON file

We start by loading the data from a JSON file. We're going to make this easy on ourselves by using a library that does most of the heavy lifting for us. To use the **json** library, we first must import it:

```
8    import json
```

This links the json library into our project so we can access its features. Now we can use it to load the file:

```
39   try:
40       file = open(FILE_NAME, "r")
41       students = json.load(file)
42       file.close()
43       print(f">>> Loaded {len(students)} records.")
```

Figure 4: Reading a JSON file using the json library.

Line 40 opens the file in read mode, and line 41 reads the contents of the file into the **students** list. We're using the **json.load()** method which handles all the work of stepping through the file and putting the data into **students** for us. We then close the file and report to the user how many records we found.

Error Handling

Handling file not found

We don't know if the file we're trying to open exists or not. If we try to operate on a non-existent file, Python will throw an exception. That's kind of an ugly experience for the user, so we're going to try to detect the existence of the file first. To do that, we'll enclose the code for opening the file inside a **try:** block. The code inside this block will attempt to load the file and offer warnings to the user if

any issues are encountered. If an exception occurs inside the **try:** block, we can create an exception handler for it. This lets us tell the user what the problem was and handle it more gracefully than just spewing an error at them.

```
39      try:
40          file = open(FILE_NAME, "r")
```

Line 39 starts the **try:** block, and line 40 tells Python to try opening a file named **Enrollments.json** in read mode. If the file does exist, line 40 assigns a handle to the file to the **file** variable and continues execution of the code within the **try:** block. If instead the file is not found, execution will jump to the **except** block on line 51 and report the error to the user. The rest of the code inside the **try:** block will be skipped.

```
50      # Let the user know we couldn't find the file
51      except FileNotFoundError:
52          print(f">>> {FILE_NAME} not found. A new file will be created.")
53          file = open(FILE_NAME, "w")
54          file.close()
```

Figure 5: Handling file not found.

Line 52 reports that the file could not be found, and that a new file will be created. Line 53 creates a new empty file, and line 54 closes the file which writes it to the file system.

Validating data in the JSON file

Once the file has been read, we should check to see if it has the data we expect inside of it. To do that, we're going to validate that the keys we're expecting (**FirstName**, **LastName**, and **CourseName**) all exist for each item in the list.

```
45      for index, item in enumerate(students):
46          # Check to see if the keys we are expecting exist in the data
47          if not all(key in item for key in KEYS):
48              raise Exception(f">>> Missing expected key at index {index}. Please check {FILE_NAME} for errors.")
```

Figure 6: Validating the JSON data.

Line 45 sets up a for loop that will step through each item in the **students** list. Line 47 looks at the current item and checks to see if all the keys we are expecting to find (as defined by the **KEYS** list) are actually present in the data we've read. If we don't find all of the keys we expect to find, line 48 raises an exception and assigns a custom error message to it that we can print later in our exception handler. Otherwise, the loop continues to run until all the items in **students** are validated.

```
56    # Let the user know some other problem occurred when loading the file
57    except Exception as e:
58        print(F"There was an error loading the data from {FILE_NAME}. Please check {FILE_NAME} and try again.")
59        print(e, e.__doc__)
60        exit()
```

Figure 7: The exception handler.

Line 57 sets up an exception handler for any other exceptions (aside from the FileNotFoundError that we already created). If any exceptions occur in our **try:** block, Python will jump to this exception handler. Line 58 presents our custom error message associated with the exception. Line 59 presents the error message that was passed to us by whatever caused the exception.

```
>>> Loading data from Enrollments.json
>>> Loaded 4 records.
>>> There was an error loading the data from Enrollments.json. Please check Enrollments.json and try again.
>>> Missing expected key at index 1. Please check Enrollments.json for errors. Common base class for all non-exit exceptions.
```

Figure 8: Example output showing what happens when the data in the JSON file is not what we expect.

As an example, if our JSON file doesn't contain all the keys we're expecting to find, the exception handler will present us with a message letting us know the JSON file is not what we expected.

The last piece of our **try:** block is the **finally:** statement, which always runs after an exception has been handled by one of our exception handlers.

```
62    # If the file is still open for some reason, close it
63    finally:
64        if not file.closed:
65            print(">>> Closing file.")
66            file.close()
```

Figure 9: Closing the file as a last step in the exception handler.

In this case, we'll just make sure that we close the file in case it got left open because some kind of error occurred when it was being read. Remember, if something goes wrong in line 41 when we're loading the json data, an exception will occur and line 42 (which closes the file) will never be executed. So the right thing to do is to close it here after the exception is handled.

Line 64 checks to see if the file is closed. If it is not, line 66 closes it.

Main Menu Loop

Now that the data is loaded, we enter the main menu loop. The setup of the main menu loop is identical to last week's exercise.

```
70  while True:
71      # Present the menu of choices
72      print(MENU)
73      menu_choice = input("Enter your choice: ")
```

Figure 10: Setting up the main menu loop.

Option 1: Register a student for a course

This week we want to validate the data as the user enters it. We only want to accept alphabetical characters for student names.

```
75      if menu_choice == '1':
76          # Input user data, allow only alpha characters
77          while True:
78              try:
79                  student_first_name = input("Enter student's first name: ")
80                  if not student_first_name.isalpha():
81                      raise ValueError(f">>> Please use only letters. Try again.\n")
82                  else:
83                      break
84              except ValueError as e:
85                  print(e)
```

Figure 11: Validating user input.

Instead of just asking for the input and storing whatever the user types, this time we'll create a little while loop which will keep asking the user for the data until it gets a valid answer. Line 77 sets up the while loop and line 79 asks the user to input data (inside a **try:** block as we did when loading the file). Line 80 checks to see if the value doesn't contain only alphabetical characters. If it doesn't, line 81 raises an exception with a custom error message. Once we receive valid data, the while loop breaks on line 83. Line 84 handles the exception if they user entered bad data, and presents the error message.

We repeat the same validation pattern with the student's last name.

```
87          while True:
88              try:
89                  student_last_name = input("Enter student's last name: ")
90                  if not student_last_name.isalpha():
91                      raise ValueError(">>> Please use only letters. Try again.\n")
92                  else:
93                      break
94              except ValueError as e:
95                  print(e)
```

Figure 12: Validating last name.

Lastly we ask the user for the course name:

```
97              course_name = input("Enter the course name: ")
```

No data validation occurs here.

Now that we have good data in our variables, we can create a list containing that data. We'll use the **student_data** dictionary to store it.

```
99          # Create dictionary using captured data
100         student_data = {"FirstName":student_first_name,"LastName":student_last_name,"CourseName":course_name}
```

Dictionaries are assigned values by enclosing the values in curly brackets. Each value contains a key-value pair formatted as ***keyName:valueName***. Multiple values are separated with commas as in line 100.

Finally, we'll append the dictionary we just created (which represents a single row of our table) to the list variable **students**.

```
102         # Append student data to students list
103         students.append(student_data)
104         print(f">>> Registered {student_first_name} {student_last_name} for {course_name}.\n")
105         saved = False  # Set the saved flag to false, so we can remind user to save
106         continue
```

Figure 13: Appending data to the students list.

Line 103 uses the **.append()** method to append the dictionary **student_data** to the end of the list **students**.

Line 104 echoes the entered data back to the user.

Line 105 sets our **saved** flag to **False** so that later on we can remind the user to save before exiting the program.

Line 106 instructs the while loop block we're in (the main menu loop) to continue from the top.

Option 2: Show current data

This week we'll present the on-screen data to the user in a more human friendly format.

```
108     elif menu_choice == '2':
109         # Display the data in a human-friendly format
110         print(">>> The current data is:\n")
111         print("First Name          Last Name           Course Name         ")
112         print("------------------------------------------------------------")
113         for item in students:
114             print(f"{item['FirstName'][:20]:<20}{item['LastName'][:20]:<20}{item['CourseName'][:20]:<20}")
115         print("------------------------------------------------------------")
116         continue
```

Figure 14: Presenting the data in a human friendly format.

Since we're emulating a table with our 2-dimensional list, let's print the data on the screen in a table-like format. We'll use a fixed width of 20 characters for each column. To begin, lines 110 – 112 display a header so we know what the data in our table is supposed to be.

Line 113 sets up a **for** loop that iterates through the list **students**. Each time through the loop we'll get a new instance of **student_data**, which contains the first name, last name, and course name data. We'll pull those values out of the dictionary using their keys.

Line 114 prints those elements to the screen, using the **[:20]:<20** string formatting notation. Let's unpack that. The **[:20]** slices the value down to 20 characters so it fits neatly under our header if it is too long. Then, the **:<20** formats the data to be left-justified, and if our data is shorter than 20 characters, spaces will be added to pad the output up to 20 characters.

Line 116 instructs the while loop block we're in to continue from the top.

```
First Name          Last Name           Course Name
------------------------------------------------------------
Luke                Skywalker           Droid Repair
Han                 Solo                Scoundreling
Leia                Organa              Scoundrel Defense
C3                  PO                  Cyborg Relations
R2                  D2                  Starfighter Repair
Darth               Vader               Advanced Villainry
```

Figure 15: Example human readable output for Option 2.

Option 3: Save the data to a file

If the user enters a value of "3", we want to save the data to a JSON file.

To do that, we'll again lean on the **json** library to dump the **students** data to a file.

```
118     elif menu_choice == '3':
119         # Save the data to a file
120         try:
121             file = open(FILE_NAME, 'w')
122             json.dump(students, file, indent=4)
123             file.close()
124             saved = True
125             print(f">>> Wrote registration data to filename {FILE_NAME}\n")
126 
127             # Print JSON data to terminal
128             json_data = json.dumps(students, indent=4)
129             print(json_data)
130 
131         except Exception as e:
132             print(">>> There was an error writing the registration data.")
133             print(f">>> {e}", e.__doc__)
134         finally:
135             file.close()
136 
137         continue
```

Figure 16: Creating the JSON file.

Line 121 opens the file with write access as we've done before, inside of a **try:** block so we can perform error handling.

Line 122 uses the **json.dump()** method to write the contents of **students** to the **file**. We also use the optional statement **indent=4** which writes the data to a file in a more human-readable format. Basically, it indents each dimension of the list by 4 spaces and puts each value on its own line. If we don't use the **indent** command, everything just gets dumped to a single line of text which is very hard to work with if a human ever needs to open the file. We then close the file on line 123.

Line 124 sets the **saved** variable to True. We'll use this later to see if we need to remind the user to save their work.

Line 125 lets the user know we saved the data to the file. We then print the data to the screen on lines 128 and 129. We use the **json.dumps()** method to do this. It works the same as **json.dump()** but allows us to put the data into a variable (**json_data**) instead of writing it to a file. We also include the **indent=4** to make it easier to read on-screen.

Lastly we have some exception handling on lines 131 through 135.

Figure 17: Example JSON file output showing human-friendly formatting resulting from indent=4

Option 4: Exit the program

We're almost at the finish line. We're going to add a quality-of-life feature when the user asks to exit the program. If they haven't saved their work, we'll remind them of this and ask if they are sure they want to exit the program.

```
139     elif menu_choice == '4':
140         # Exit if data has already been saved or was unmodified (i.e. saved = undefined)
141         if (saved is False):
142             exit_confirm = input(">>> File not saved. Are you sure you want to exit? (Y/N): ")
143             if exit_confirm.capitalize() == 'Y':
144                 print(">>> Have a nice day!\n")
145                 break
146             else:
147                 continue
148         else:
149             print(">>> Have a nice day!\n")
150             break
```

Figure 18: Warning the user that their data hasn't been saved.

Line 141 uses an **if** statement to check if **saved** is **False**. Remember we set that to **False** back in option 1 (line 105) when the user added new data. If they haven't saved, we spin up another **if** statement on line 143 to ask them if they are sure they want to exit. We format the user's response using the **.capitalize()** method to make sure we detect both lower and upper case Y.

```
------ 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 choice: 4
>>> File not saved. Are you sure you want to exit? (Y/N):
```

Figure 19: Example output of exit confirmation

If they enter Y the main menu **while** loop will exit via the break statement on line 145.

If they enter any other value, line 147 executes which goes back to the top of the main menu **while** loop.

If **saved** evaluates to True, we skip all this stuff and just encourage the user to have a nice day. The break on line 150 exits the main menu **while** loop.

Invalid Option

If the user selects an invalid option from the main menu, we ask them to try again.

```
152         else:
153             print("Please only choose option 1, 2, 3, or 4.")
```

Summary

In this assignment we learned:

1. How to use dictionaries to store data
2. How to use lists-of-dictionaries (2-dimensional lists) to store data in a table-like format
3. How to use external libraries to make our life easier
4. How to parse data from JSON files into lists and dictionaries
5. How to process data in lists and dictionaries, and display them to the screen
6. How to generate a JSON file from data in lists and dictionaries
7. How to perform data validation
8. How to perform error handling