Caitlyn McFarland

February 26, 2025

IT FDN 110 A

Assignment 05

GitHub Link: https://github.com/CaitMcF/IntroToProg-Python-Mod05

Advanced Collections and Error Handling

Intro

Using the script from Assignment 4, this week focused on using JSON files (JavaScript Object Notation) and using error handling messages to ensure data integrity within your code. Since JSON files use dictionaries rather than lists, the variable set up was slightly different. Dictionaries use key-value pairs where each value is a unique key. This is helpful for creating tables with data sets. JSON is an easy to use format that allows for storing and organizing data and can be readily used between different applications.

Error Handling

Structured error handling is useful in writing scripts because it can prevent the user from entering data in an incorrect format. It can also make sure data is saved appropriately. In this case the first example of this is seen in Figure 1. The 'FileNotFoundError' will let the user know that there is not a file with the name that you are attempting to open (Figure 2) and then creates the file that you designated earlier.

Error Handling was also used for the input options in the menu. For the first menu choice (Figure 3) the error handling added in made it so that the input values for the first and last name had to be letters, or it would send an error code. The third menu choice also had error handling (Figure 4). The try-except construct allows you to direct the user to resolve the problem with messages when data is entered that does not match the desired values.

```
try:
    file = open(FILE_NAME, "r")
    students = json.load(file)
    file.close()
except FileNotFoundError as e:
    print("File not found. Please create file")
    print("-- Technical Error Message -- ")
    print(e, e.__doc__, type(e), sep='\n')
    file = open(FILE_NAME, 'w')
    json.dump(students, file)
except Exception as e:
    print("There was a non-specific error!\n")
    print("-- Technical Error Message -- ")
    print(e, e.__doc__, type(e), sep='\n')
finally:
    if not file.closed:
        print("Closing file")
        file.close()
```

Figure 1: Error Handling. When the script starts, there is structured error handling in place to ensure that a file exists to write into.

```
File not found. Please create file
-- Technical Error Message --
[Errno 2] No such file or directory: 'Enrollments.json'
File not found.
<class 'FileNotFoundError'>
Closing file
```

Figure 2: Error Message. When the script starts without a file to write to, it will display an error message and then create a file after it closes.

```python
# Input user data
if menu_choice == "1":  # This will not work if it is an integer!
    try:
        student_first_name = input("Enter the student's first name: ")
        if not student_first_name.isalpha():
            raise ValueError("The first name should only contain letters.")
        student_last_name = input("Enter the student's last name: ")
        if not student_last_name.isalpha():
            raise ValueError("The last name should only contain letters.")
        course_name = input("Please enter the name of the course: ")
        student_data = {"FirstName": student_first_name,
                        "LastName": student_last_name,
                        "Course": course_name}
        students.append(student_data)
        print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
        continue
    except ValueError as e:
        print(e)
        print("-- Technical Error Message -- ")
        print(e.__doc__)
        print(e.__str__())
    except Exception as e:
        print("There was a non-specific error!\n")
        print("-- Technical Error Message -- ")
        print(e, e.__doc__, type(e), sep='\n')
    continue
```

Figure 3: Error Handling with input data. This structured error handling allows for data to be input as the writer would like to store in the file.

```python
# Save the data to a file
elif menu_choice == "3":
    try:
        file = open(FILE_NAME, "w")
        json.dump(students, file)
        file.close()
        print("The following data was saved to the file:")
        for student in students:
            print(f"Student {student["FirstName"]} {student["LastName"]} "
                  f"is enrolled in {student["Course"]}")
        continue
    except TypeError as e:
        print("Please check that the data is in a valid JSON format\n")
        print("-- Technical Error Message -- ")
        print(e, e.__doc__, type(e), sep='\n')
    except Exception as e:
        print("-- Technical Error Message -- ")
        print("Built-In Python error info: ")
        print(e, e.__doc__, type(e), sep='\n')
    finally:
        if not file.closed:
            file.close()
```

Figure 4: Error Handling with input data. This structured error handling allows for data to be input as the writer would like to store in the file.

Testing the code in PyCharm and the command line

Figures 5 and 6 show the script working in PyCharm and in the command line. The output for both creates a compatible format for JSON files. Below shows the input data with the original data that was in the file.


```
[{"FirstName": "Sam", "LastName": "Smith", "Course": "Python 100"},
{"FirstName": "Teri", "LastName": "James", "Course": "Python 200"},
{"FirstName": "Vic", "LastName": "Vu", "Course": "Python 200"},
{"FirstName": "Susan", "LastName": "Samuel", "Course": "Python 100"},
{"FirstName": "Hailey", "LastName": "Mae", "Course": "Python 200"},
{"FirstName": "Tim", "LastName": "True", "Course": "Python 500"}]
```


Figure 5. PyCharm output in .json file format

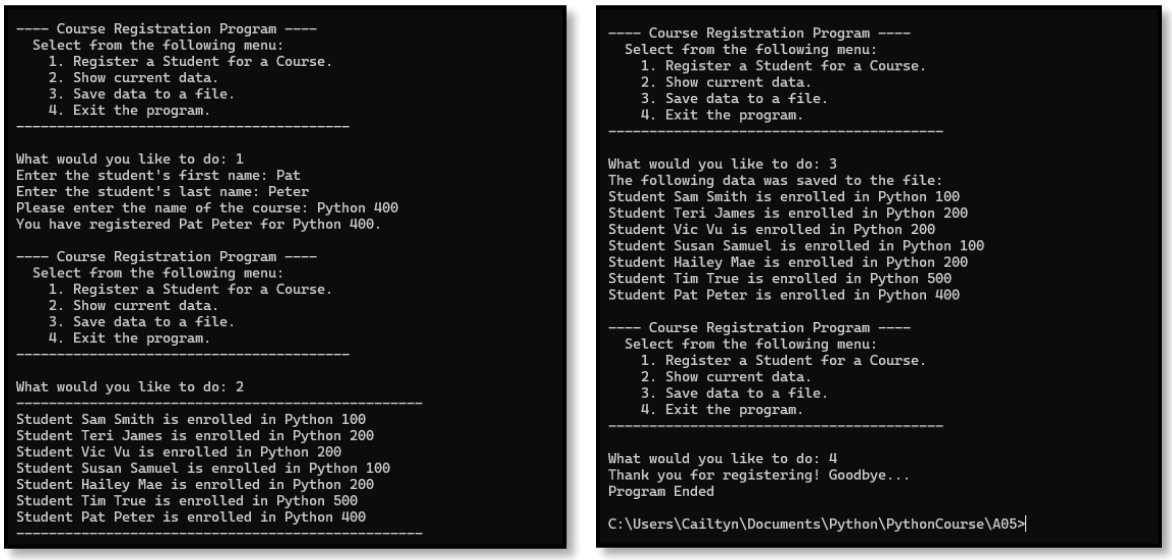

Figure 6: Running in the command line.

Summary

Advanced collections and error handling techniques allow for an easy way to store and handle data to ensure that it is properly entered and saved. If errors are managed effectively and dictionaries are used to provide a flexible way to store data, your script can be run by many different users on multiple different platforms without impact to your original script.