Nathan Parece

5/21/2025

IT FDN 110 A

github.com/Nathan-Parece/IntroToProg-Python-Mod05

Dictionaries, Errors, and Git

Introduction

This week, we introduced dictionaries and explored structured error handling and version control. Dictionaries are like lists, but instead of using indices to identify elements, they use keys. They're also natively in the JSON format, which turns out to be very convenient for this application. We updated the existing student registration script to use dictionaries and the try-except structure for error handling. We also hosted the script in a GitHub repository so we can track new versions and easily share our code with others.

Constants, Variables, and Packages

The first change to the script happens immediately after the header: we import a module. Modules extend Python's library of functions. In general, increasing the number of built-in functions a script has access to makes it easier and more convenient to write code, but also makes the code less efficient. Modules let us pick and choose what built-in functions we want to add, balancing simplicity and efficiency. For this application, we want to work with the JSON file format, so we import the JSON module.

```
# Import Modules
import json
```

Fig. 1: Import json

The constants and variables remain almost unchanged. The FILE_NAME constant changes from Enrollments.csv to Enrollments.json, and student_data is changed from a list to a dict.

```
FILE_NAME: str = "Enrollments.json"

student_data: dict =  {} # one row of student data
```

Fig. 2: Constants and Variables

Loading a JSON File

Python opens all text files in the same way: the open() function. Just like last week, we load in read mode. Unlike last week, though, we can save a few lines of code and just use the json.load() function to get the data in Enrollments.json into the script. This way, it's already formatted and ready to go. We then close the file to keep everything neat and organized.

```
file = open(FILE_NAME, "r")
students = json.load(file)
file.close()
```

Fig. 3: Loading the File

We've also embedded this section of code in a try-except block. This lets us catch errors and handle them without crashing the script. First, we indent the lines that load the file, and put them inside of a try statement.

```
try:
    file = open(FILE_NAME, "r")
    students = json.load(file)
    file.close()
```

Fig. 4: Starting the try-except

From previous versions of this script, we know that we'll get an error if Enrollments.json doesn't exist. Python calls this a FileNotFoundError, and we can call this error out specifically.

```
except FileNotFoundError as e:
    print("Text file must exist before running this script!\n")
    print("-- Technical Error Message --")
    print(e, e.__doc__, type(e), sep="\n")
```

Fig. 5: FileNotFoundError

Everything else gets caught by the general Exception type.

```
except Exception as e:
    print("There was a non-specific error!\n")
    print("-- Technical Error Message --")
    print(e, e.__doc__, type(e), sep="\n")
```

Fig. 6: General Exception

For now, all we do with these errors is inform the user that they've occurred. We also include a block at the end to close the file. This way, if the error occurred between opening and closing the file in the initial try block, the file still gets closed.

```
finally:
    if not file.closed:
        file.close()
```

Fig. 7: finally Block

If there haven't been any errors, we can now proceed to the main loop.

Updates to the Main Loop

Much of the code in the main loop hasn't changed from last week. We're still using an if-elif-else block nested in a while loop to handle user input.

```
# Process user input, modify the data
while True:

    # Present the menu of choices
    print(MENU)
    menu_choice = input("What would you like to do: ")
```

Fig. 8: Main Loop

User Input

Option 1 has two substantial changes from last week. First, everything has been put inside of a try-except structure. Generally, this lets us restrict what data the user can input and work with, which has a wide range of uses. For this application, we're just going to make sure that students' names don't have any numbers in them. First, we check if the string input for the student's first name has any numbers in it with the isalpha() method.

```
# 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():
```

Fig. 9: Input try

The isalpha() method evaluates to True if and only if the string it acts on contains only letters. If it evaluates to False, we want to tell the user that they've made an error, so we use the raise statement to store a new ValueError with the message, "Student first names must contain only letters!"

```
raise ValueError("Student first name must contain only letters!\n")
```

Fig. 10: New ValueError

We can do the same thing when we get to the last name.

```
student_last_name = input("Enter the student's last name: ")
if not student_last_name.isalpha():
    raise ValueError("Student last name must contain only letters!\n")
```

Fig. 11: Last Name

If one of these errors is raised, the script will skip to the except blocks. Critically, these blocks happen after the lines that save the input to the students table, so that process is skipped and the bad inputs are either overwritten by the next use of option 1 or discarded when the program ends.

```
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
```

Fig. 12: Handling Exceptions

As with the try-except structure that loaded the file, we include a general Exception block to handle anything unexpected.

If an exception isn't raised, we can get the course name from the user and then format and save the inputs. The second change to this section of the code is here: rather than saving the inputs in a list, we're saving them in a dictionary. This doesn't have any immediate consequences. We also print a message telling the user informing them that they've successfully registered a student for a course.

```
course_name = input("Please enter the name of the course: ")

student_data = {"FirstName" : student_first_name,
                "LastName" : student_last_name,
                "CourseName" : course_name}
students.append(student_data)
print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
```

Fig. 13: Processing Valid Inputs

It's important to note that this section of the code has a glaring flaw: hyphenated names. Because hyphens are not letters, this program won't accept them. This is a general problem with data validation; plenty of people's names are not formatted according to the standards that one might expect. Fixing this issue is outside the scope of this assignment, but I'm looking forward to digging into it in the future.

Presenting the Data

This section has changed only slightly from last week. The starter file included a neat trick to print a long line of hyphens: apparently, strings can be multiplied by integers in the print() function. We loop over the students list of dicts and print each element in a formatted string. Now that the data is stored in a dict instead of a list, we can use keys instead of indices to select what we want to print in each step, which makes our code much more readable.

```
# Present the current data
elif menu_choice == "2":

    # Process the data to create and display a custom message
    print("-"*50)
    for student_data in students:
        print(f"FirstName : {student_data["FirstName"]}, "
        f"LastName : {student_data["LastName"]}, "
        f"CourseName : {student_data["CourseName"]}")
    print("-"*50)
    continue
```

Fig. 14: Present the Data

The print statement inside the for loop could be on just one line, but it's easier to read when it's broken up into pieces.

Saving the Data

In order to save the data, we open the Enrollments.json file in write mode, use a function to write what we've stored in the students variable, and close the file. This segment of the code has gotten much simpler! Because the data in students is already in JSON format, there's no need to change anything at this step: we just use json.dump(). The first argument is the data we want to save, and the second argument is a file object.

```
file = open(FILE_NAME, "w")
json.dump(students, file)
file.close()
print(students)
```

Fig. 15: Using json.dump()

We've also embedded this section into a try-except structure of its own. We're specifically looking out for badly-formatted data, which raises a TypeError. As with the try-except structure we used to load the file, we catch any unspecified errors using Exception and include a block at the end to close the file if it hasn't been closed already. All together, it looks like this:

```
# Save the data to a file
elif menu_choice == "3":
    try:
        file = open(FILE_NAME, "w")
        json.dump(students, file)
        file.close()
        print(students)
    except TypeError as e:
        print("Please check that the data is in 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()
    continue
```

Fig. 16: Handling Errors with File Operations

Everything else looks the same:

```
    # Stop the loop
    elif menu_choice == "4":
        break  # out of the loop
    else:
        print("Please only choose option 1, 2, 3, or 4")

print("Program Ended")
```

Fig. 17: Unchanged Code

Testing

Testing this week proceeds similarly to previous versions. We'll need to test each menu option and check if the data is saved correctly. Now that we have error handling, though, we can also deliberately cause errors and see if the program reacts how we expect it to. I tested using PyCharm first and Command Prompt second, but for the sake of readability I've combined the testing screenshots from each. Any differences between the two will be noted when they occur.

At runtime, the first thing that our script tries to do is open Enrollments.json. If this file doesn't exist, we get this error:

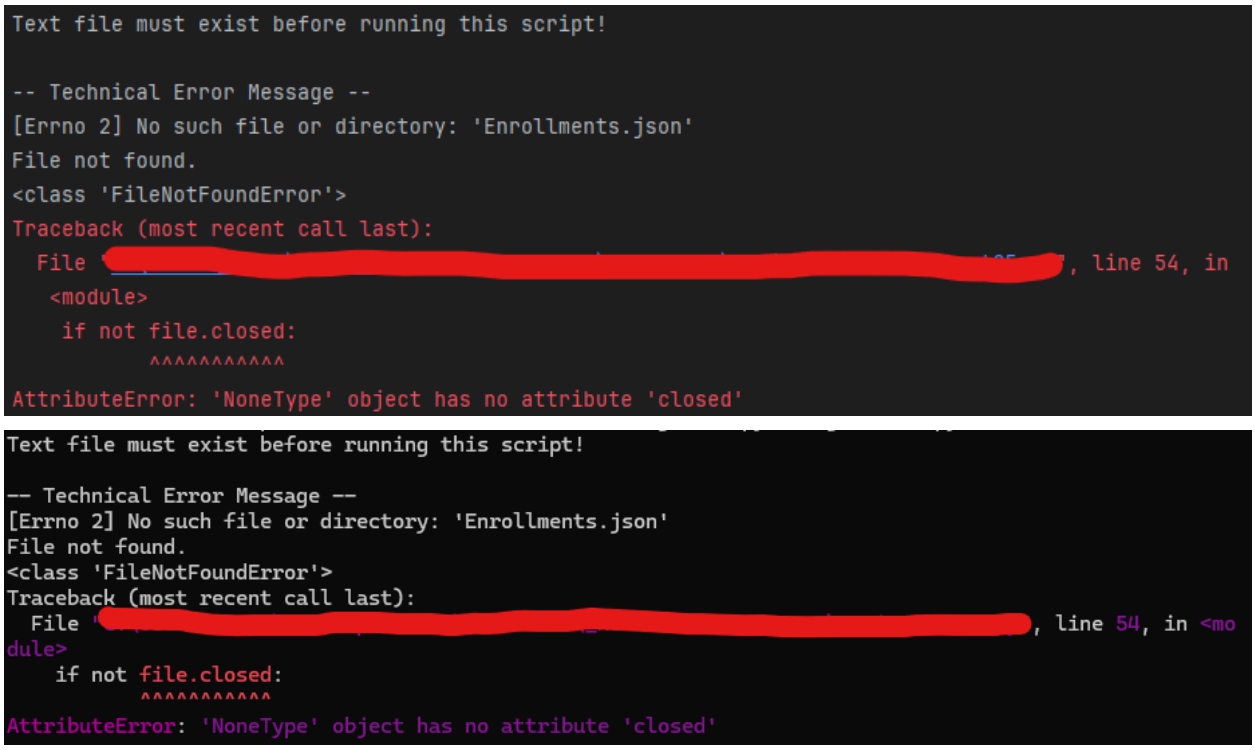
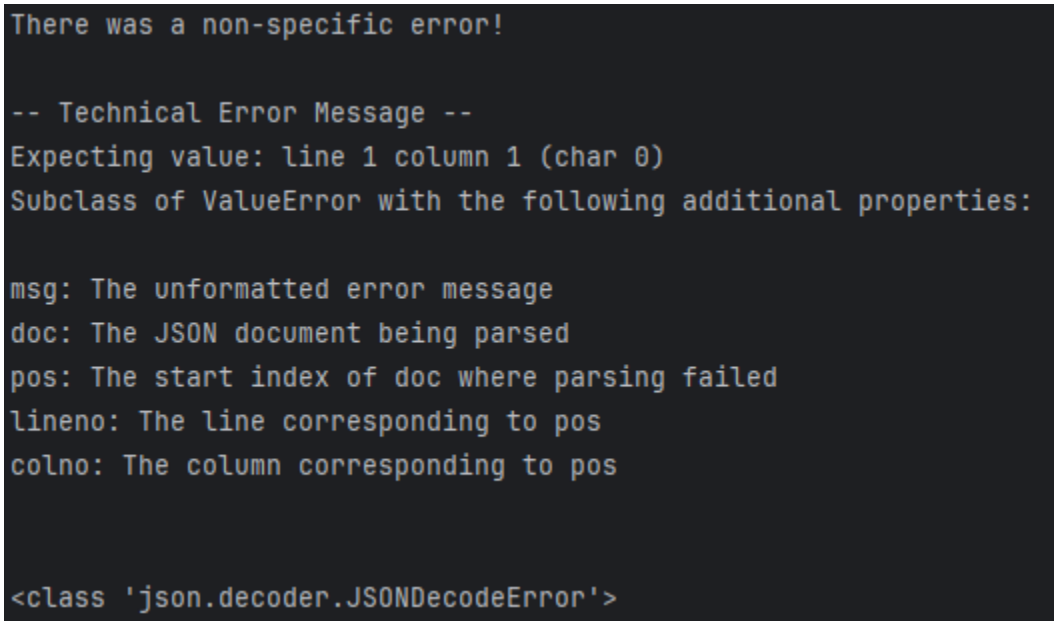


Fig. 18: No Such File

Likewise, if the file data isn't in JSON format, we get this error:



```
There was a non-specific error!

-- Technical Error Message --
Expecting value: line 1 column 1 (char 0)
Subclass of ValueError with the following additional properties:

msg: The unformatted error message
doc: The JSON document being parsed
pos: The start index of doc where parsing failed
lineno: The line corresponding to pos
colno: The column corresponding to pos


<class 'json.decoder.JSONDecodeError'>
```

Fig. 19: Junk Data

I caused this error by opening Enrollments.json in a text editor and appending the string “junk data” to the start of the file. Concerningly, this error doesn’t cause the program to halt! Because the data hasn’t been loaded into the program’s memory, if we were to save the file at this point, the data in the file would be lost. We could fix this by adding a new except block like this:

```
except json.decoder.JSONDecodeError as e:
    print("Text file must be in JSON format!\n")
    print("-- Technical Error Message --")
    print(e, e.__doc__, type(e), sep="\n")
    quit()
```

Fig. 20: Potential Fix

This is safer for the existing data. However, it wasn’t included in the assignment prompt, so I’ve elected not to include it in my submission.

Running the script with a correctly-formatted Enrollments.json file brings us to the menu:

```
---- 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:
```

```
---- 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:
```

Fig. 21: Program Loads Correctly

Let's try option 1 with a valid name format. I'll register myself in PyCharm and Alice Roberts in the Command Prompt.

```
What would you like to do: 1
Enter the student's first name: Nathan
Enter the student's last name: Parece
Please enter the name of the course: Python 100
You have registered Nathan Parece for Python 100.
```

```
What would you like to do: 1
Enter the student's first name: Alice
Enter the student's last name: Roberts
Please enter the name of the course: Python 100
You have registered Alice Roberts for Python 100.
```

Fig. 22: Valid Inputs

Next, let's test an invalid first name: N4than.

```
What would you like to do: 1
Enter the student's first name: N4than
Student first name must contain only letters!

-- Technical Error Message --
Inappropriate argument value (of correct type).
Student first name must contain only letters!
```

```
What would you like to do: 1
Enter the student's first name: N4than
Student first name must contain only letters!

-- Technical Error Message --
Inappropriate argument value (of correct type).
Student first name must contain only letters!
```

Fig. 23: Invalid First Name

Our error handling caught this error and took us back to the menu without adding anything to the students table. Next, let's try an invalid last name: Nathan Par3ce.

```
What would you like to do: 1
Enter the student's first name: Nathan
Enter the student's last name: Par3ce
Student last name must contain only letters!

-- Technical Error Message --
Inappropriate argument value (of correct type).
Student last name must contain only letters!
```


```
What would you like to do: 1
Enter the student's first name: Nathan
Enter the student's last name: Par3ce
Student last name must contain only letters!

-- Technical Error Message --
Inappropriate argument value (of correct type).
Student last name must contain only letters!
```


Fig. 24: Invalid Last Name

We get the same result as when we entered an invalid first name. Let's look at the data we have so far:

```
What would you like to do: 2
-------------------------------------------------
FirstName : Bob, LastName : Smith, CourseName : Python 100
FirstName : Sue, LastName : Jones, CourseName : Python 100
FirstName : Nathan, LastName : Parece, CourseName : Python 100
-------------------------------------------------
```


```
What would you like to do: 2
-------------------------------------------------
FirstName : Bob, LastName : Smith, CourseName : Python 100
FirstName : Sue, LastName : Jones, CourseName : Python 100
FirstName : Nathan, LastName : Parece, CourseName : Python 100
FirstName : Alice, LastName : Roberts, CourseName : Python 100
-------------------------------------------------
```


Fig. 25: Displaying Data

Note that because I tested with Pycharm first, the data from that session is present in the Command Prompt session. This confirms that the script is saving our data correctly, but we'll check the JSON file at the end anyway, just to make sure.

Now let's save our data to a file:

```
What would you like to do: 3
[{'FirstName': 'Bob', 'LastName': 'Smith', 'CourseName': 'Python 100'}, {'FirstName': 'Sue',
 'LastName': 'Jones', 'CourseName': 'Python 100'}, {'FirstName': 'Nathan', 'LastName': 'Parece',
 'CourseName': 'Python 100'}]
```

```
What would you like to do: 3
[{'FirstName': 'Bob', 'LastName': 'Smith', 'CourseName': 'Python 100'}, {'FirstName': 'Sue', 'LastName
': 'Jones', 'CourseName': 'Python 100'}, {'FirstName': 'Nathan', 'LastName': 'Parece', 'CourseName': '
Python 100'}, {'FirstName': 'Alice', 'LastName': 'Roberts', 'CourseName': 'Python 100'}]
```

Fig. 26: Saving Data

The formatting is a little ugly, but that's what it's supposed to look like. The code behind option 4 and the last else clause hasn't changed, but we can test it anyway. Here's option 4:

```
What would you like to do: 4
Program Ended
```

```
What would you like to do: 4
Program Ended
```

Fig. 27: End the Program

And here's what happens when we input something that isn't an option:

```
What would you like to do: test
Please only choose option 1, 2, 3, or 4
```

```
What would you like to do: test
Please only choose option 1, 2, 3, or 4
```

Fig. 28: Invalid Menu Choice

We can also test more of our error handling code here! As far as I know, we can't get the data into a bad format with just user input, but we can cause an error by putting Enrollments.json into read-only mode. When we try option 3, this is what happens:

```
What would you like to do: 3
-- Technical Error Message --
Built-In Python error info:
[Errno 13] Permission denied: 'Enrollments.json'
Not enough permissions.
<class 'PermissionError'>
```

```
What would you like to do: 3
-- Technical Error Message --
Built-In Python error info:
[Errno 13] Permission denied: 'Enrollments.json'
Not enough permissions.
<class 'PermissionError'>
```

Fig. 29: Permissions Error

Finally, we can take a look at Enrollments.json after all of our testing is done.

```
[{"FirstName": "Bob", "LastName": "Smith", "CourseName": "Python
100"}, {"FirstName": "Sue", "LastName": "Jones", "CourseName":
"Python 100"}, {"FirstName": "Nathan", "LastName": "Parece",
"CourseName": "Python 100"}, {"FirstName": "Alice", "LastName":
"Roberts", "CourseName": "Python 100"}]
```


Fig. 30: Enrollments.json After Testing

It's a little ugly, but this is valid JSON formatting! These tests confirm that our script works as intended.

GitHub

The last step of this assignment was to post this file, the script, and the accompanying file to GitHub. I already had a GitHub account, so this was simple and easy. We created a new repository, uploaded all of the required files, committed the changes, and posted a link to GitHub on Canvas.

Summary

The newest version of our student registration program is more user-friendly and robust. By adding in error handling, we can let users know when things go wrong and give them guidance on how to resolve these situations. The saved data is also more human-readable, with every piece of data clearly tagged.

This version is also easier for a coder to work with. Rather than using indices to select elements from a list, we now use keys, making it more obvious what each step of the various for loops is doing. If new bugs crop up, we can use our existing try-except structure to handle them. Finally, The script is also hosted on GitHub now, which will make future updates visible and improve our ability to collaborate.