Name: Jason Coult
Due: 11/13/2024
Turn-in Date: 12/2/2024
Note: Have advance permission from Instructor for late submission due to birth of my daughter
Topic: Module 05 Assignment

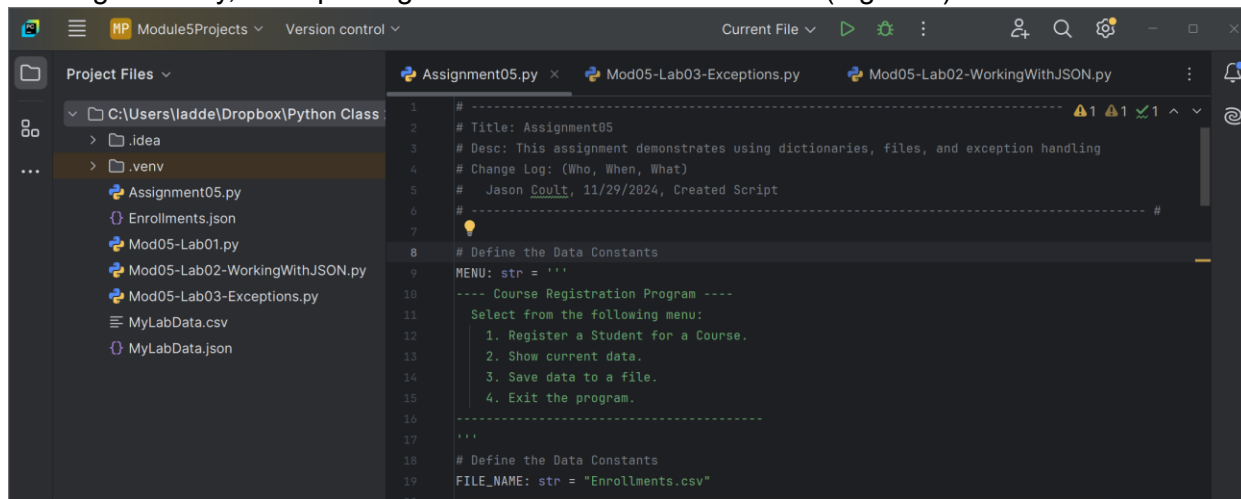# Assignment 05 – Collections & Error Handling

Introduction

This assignment introduces the use of JavaScript Object Notation (.JSON) files and error handling. The ability to use .JSON files is an important feature of Python, allowing transfer and storage of data following a common format and alternative to .CSV files. Error handling is essential to preventing unforeseen error (e.g. user input error) from crashing a program, allowing the programmer to anticipate and handle errors in advance in a controlled way that ensures the program can continue or prompt a user to correct the error.

This assignment is similar to Assignment 04, allowing input, manipulation, loading, and saving of data files, while incorporating error handling and use of .JSON files (instead of .CSV files) as the medium of storage.

Assignment Completion Process

Project setup

I began the assignment by renaming the Assignment05 starter .py file, adding it to my Module 5 working directory, and updating the header block within the file (Figure 1).

Figure 1: Project setup

Next I edited the start file's constants and variables to match those required by the assignment specifications. A notable addition to the variables is the *student_data* dictionary. I temporarily commented out the *csv_data* string for now, as it may not be needed (Figure 2).

```
# Define the variables
student_first_name: str = ''   # Holds the first name of a student entered by the user.
student_last_name: str = ''    # Holds the last name of a student entered by the user.
course_name: str = ''   # Holds the name of a course entered by the user.
student_data: dict = {}   # dictionary of student data
students: list = []   # a table of student data
#csv_data: str = ''   # Holds combined CSV data.
file = None   # Holds a reference to an opened file.
menu_choice: str   # Hold the choice made by the user.
```

Figure 2: Variable initialization

Program start-up processing

First, the program is required to load "Enrollments.json" into a 2-D list of dictionary rows on startup. Following the example of loading a .json file from Lab02, I attempted to load the file contents into the *students* variable using the json.load() method to open the file object. However, when I ran the program (in debug mode), the program crashed with a NameError message (Figure 3).
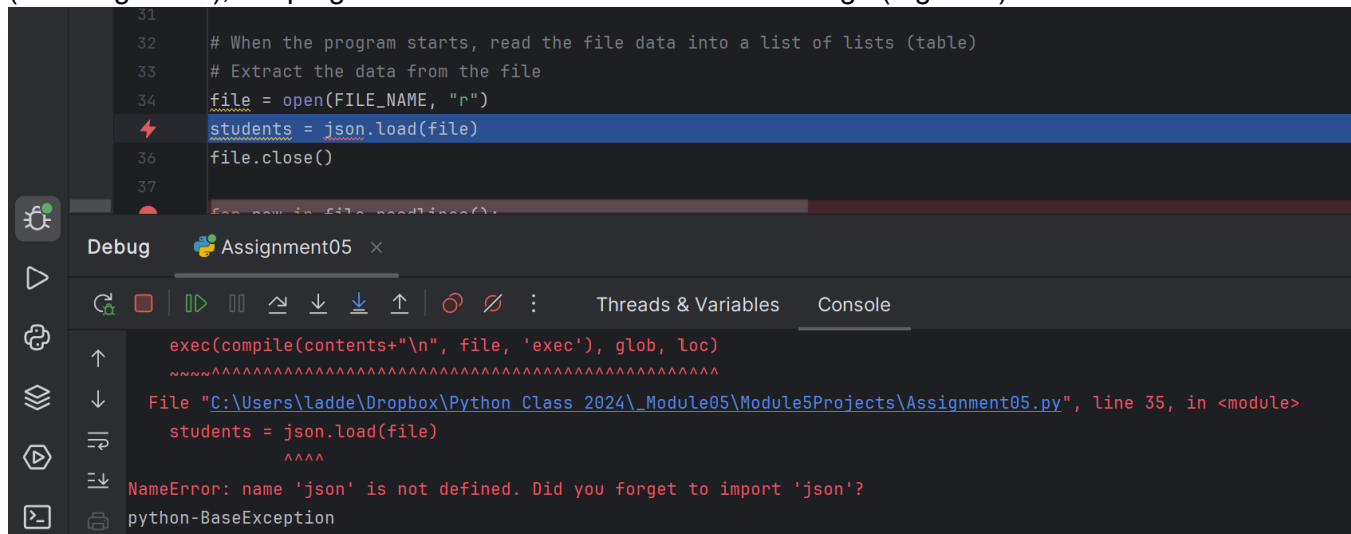

Figure 3: Attempt to load .json file

After examining my Lab02 project, I realized that this error results from not loading the json Python library at the start of the program. I added "import json" at the top of the script and ran the program again, and confirmed that indeed the .json file contents had been loaded as a list of dictionary rows into the *students* variable (Figure 4).

```
34    # When the program starts, read the file data into a list of lists (table)
35    # Extract the data from the file
36    file = open(FILE_NAME, "r")
37    students = json.load(file)
38    file.close()
39
●     for row in file.readlines():
```

bug   Assignment05 ×

■ ▶ ⏸ ⌃ ↓ ↓ ↑ | ⊘ ⊘ ⋮   Threads & Variables   Console

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
<n    10 course_name = {str} "
      01
   >  ▤ file = {TextIOWrapper} <_io.TextIOWrapper name='Enrollments.json' mode='r' encoding='cp1252'>
   >  ▤ student_data = {dict: 0} {}
      10 student_first_name = {str} "
      01
      10 student_last_name = {str} "
      01
   >  ▤ students = {list: 2} [{'CourseName': 'Python 100', 'FirstName': 'Bob', 'LastName': 'Smith'}, {'CourseName': 'Python 100', 'FirstName': 'Sue', 'LastName':
   >  ▦ Special Variables
```

Figure 4: Confirming that .json file loads into list of dictionary entries

To ensure the file was loaded correctly, I also confirmed the contents of the .json file (created when completing lab exercises) (Figure 5).

📝 Enrollments.json   ×   +

File   Edit   View

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

Figure 5: .JSON file contents

This confirms that the startup requirement to load the file is met. I removed the prior code from the starter file which loaded data from .csv using a loop. The .json method is certainly much more elegant.

Menu option 1

Menu choice 1 accepts user input and adds the data to *students*. The starter code collects string inputs but then does not properly format them to append to a dictionary list. First, I commented out the starter code's original assignment of a single student's data to *student_data*. Then, using the appropriate key variable names, and following the example in Lab02, I added code to create a single students' data to a dictionary row assigned to *student_data*. I confirmed this worked by running the debugger, entering a third student Johnny Smith, and verifying it was successfully appended to the *students* variable (Figure 6).

```
 46
 47        # Input user data
 48        if menu_choice == "1":  # This will not work if it is an integer!
 49            student_first_name = input("Enter the student's first name: ")
 50            student_last_name = input("Enter the student's last name: ")
 51            course_name = input("Please enter the name of the course: ")
 52            #student_data = [student_first_name,student_last_name,course_name]
 53            student_data = {"CourseName": course_name,
 54                            "FirstName": student_first_name,
 55                            "LastName": student_last_name}
 56            students.append(student_data)
 57            print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
            continue
 59
```

Assignment05 ×

Threads & Variables   Console

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
course_name = {str} 'English101'
file = {TextIOWrapper} <_io.TextIOWrapper name='Enrollments.json' mode='r' encoding='cp1252'>
menu_choice = {str} '1'
student_data = {dict: 3} {'CourseName': 'English101', 'FirstName': 'Johnny', 'LastName': 'Smith'}
student_first_name = {str} 'Johnny'
student_last_name = {str} 'Smith'
students = {list: 3} [{'CourseName': 'Python 100', 'FirstName': 'Bob', 'LastName': 'Smith'}, {'CourseName': 'Python 100',
    0 = {dict: 3} {'CourseName': 'Python 100', 'FirstName': 'Bob', 'LastName': 'Smith'}
    1 = {dict: 3} {'CourseName': 'Python 100', 'FirstName': 'Sue', 'LastName': 'Jones'}
    2 = {dict: 3} {'CourseName': 'English101', 'FirstName': 'Johnny', 'LastName': 'Smith'}
    len = {int} 3
```

Figure 6: Confirming menu option 1 (data input)

This confirms that menu option 1 functions correctly. I proceeded to remove the commented line and continue.

Menu option 2

The second menu option simply presents the data contained in *students* in the console. First, I simply ran the starter code as-is for option 2 but it crashed as expected (Figure 7).

Figure 7: Starter code for menu option 2 crashes

Students is a list of dictionary entries. The week's notes indicate that dictionary entries are accessed with a syntax of row["Key"]. The current starter code has row[number] syntax (left over from accessing array entries). Therefore I simply replaced the starter code array values with the key name (proper way to access dictionary entry) in place of the index value (used previously for accessing array entry). Running the debugger confirmed that the data was indeed correctly displayed (Figure 8).

```
 57
 58            # Present the current data
 59            elif menu_choice == "2":
 60
 61                # Process the data to create and display a custom message
 62                print("-"*50)
 63                for student in students:   student: {'CourseName': 'Python 100', 'FirstName': 'Sue', 'LastName': 'Jones'}
 64                    print(f"Student {student["FirstName"]} {student["LastName"]} is enrolled in {student["CourseName"]}")
 65                print("-"*50)
 ●                  continue
 67
```

ebug   🐍 Assignment05 ×

■ ❚▷ ❚❚ △ ↓ ↧ ↥ ⊘ ∅ ⋮   Threads & Variables   Console

```
---- 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: >? 2
------------------------------------------------
Student Bob Smith is enrolled in Python 100
Student Sue Jones is enrolled in Python 100
------------------------------------------------
```

Figure 8: Confirm menu option 2 functionality

Having confirmed that menu option 2 functions correctly, I proceeded to option 3.

Menu option 3

The third menu choice requires that all *students* data be written to a .json file and then the *students* data is again displayed, similar to option 2. Instead of looping through each row of *students* and writing the text to a .csv file, as in the prior assignment, the json.dump() function (as shown in the Lab02 example) allows a properly-formatted variable to be written in a single line.

I retained the original starter code for menu option 3 but commented out the loop that traversed the list of lists from Assignment 4. I replaced the loop with a single line using json.dump(). To test this, I ran the program and entered a third student manually using menu option 1; then I saved the data to the .json file using menu option 3 and stopped the code with a breakpoint right after the line that closed the file, confirming that (a) my student entry had successfully been added to *students*, and (b) the .json file had been updated with the new student (Figure 9).
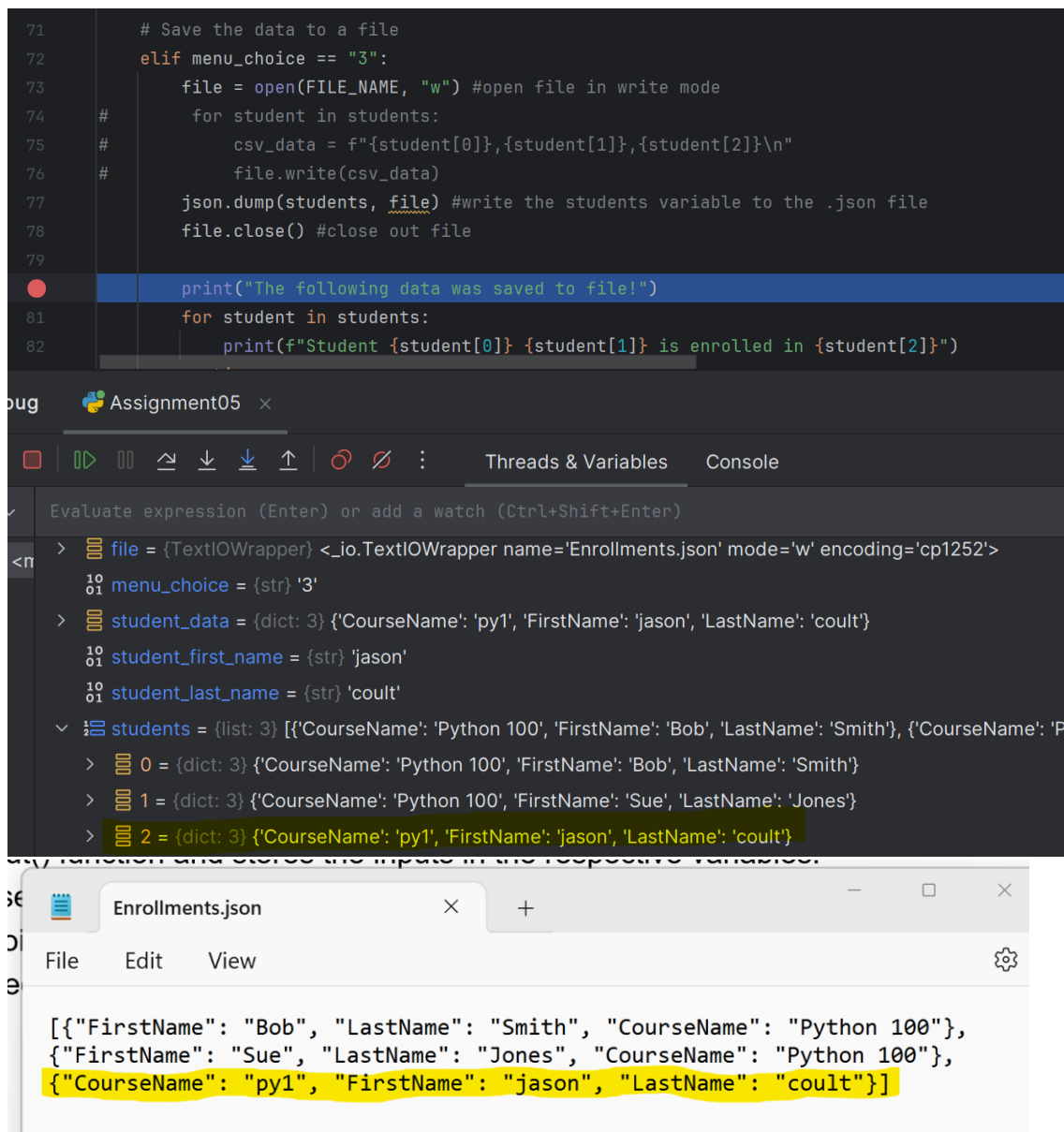
```
71          # Save the data to a file
72          elif menu_choice == "3":
73              file = open(FILE_NAME, "w") #open file in write mode
74    #           for student in students:
75    #               csv_data = f"{student[0]},{student[1]},{student[2]}\n"
76    #               file.write(csv_data)
77              json.dump(students, file) #write the students variable to the .json file
78              file.close() #close out file
79
●               print("The following data was saved to file!")
81              for student in students:
82                  print(f"Student {student[0]} {student[1]} is enrolled in {student[2]}")
```


bug   🐍 Assignment05 ×

⬛ ▷▷ ⏸ ⌃ ↓ ↧ ↑ | ⊘ ∅ ⋮   Threads & Variables   Console

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

`<n` > ☰ file = {TextIOWrapper} <_io.TextIOWrapper name='Enrollments.json' mode='w' encoding='cp1252'>
      ¹⁰₀₁ menu_choice = {str} '3'
      > ☰ student_data = {dict: 3} {'CourseName': 'py1', 'FirstName': 'jason', 'LastName': 'coult'}
      ¹⁰₀₁ student_first_name = {str} 'jason'
      ¹⁰₀₁ student_last_name = {str} 'coult'
      ∨ ☰ students = {list: 3} [{'CourseName': 'Python 100', 'FirstName': 'Bob', 'LastName': 'Smith'}, {'CourseName': 'P
         > ☰ 0 = {dict: 3} {'CourseName': 'Python 100', 'FirstName': 'Bob', 'LastName': 'Smith'}
         > ☰ 1 = {dict: 3} {'CourseName': 'Python 100', 'FirstName': 'Sue', 'LastName': 'Jones'}
         > ☰ 2 = {dict: 3} {'CourseName': 'py1', 'FirstName': 'jason', 'LastName': 'coult'}

📄 Enrollments.json   ×   +   — □ ×

File   Edit   View   ⚙

[{"FirstName": "Bob", "LastName": "Smith", "CourseName": "Python 100"},
{"FirstName": "Sue", "LastName": "Jones", "CourseName": "Python 100"},
{"CourseName": "py1", "FirstName": "jason", "LastName": "coult"}]

Figure 9: Confirming successful save of new data (top) to .json file (bottom).

I did notice that within the .json file, the new data entries had the order of key variables reversed. However, since this is a dictionary, I believe this does not matter, since each entry is accessed by the key name and its position does not matter like it would with an array.

The next step for menu option 3 is to display the saved data. Here, I simply pasted the same lines from menu option 2 which loops through the *students* table and displays each student. I modified the code slightly to indicate this was the data that was saved. I also removed the commented-out code left over from the starter script (Figure 10).

Figure 10: Confirming display of saved data in menu option 3

Menu option 4

The fourth menu option is required to close the program. The starter script already appears to have the correct code for this, so no modification was necessary and the program exited successfully when testing option 4 (Figure 11).

Figure 11: Menu option 4

Catch-all if user selects invalid menu option

The starter code had a catch-all "else" statement at the end of the while loop, in case the user did not select a valid menu choice; however, the display message was incorrect, which I modified to include a prompt to select a valid option of 1-4 (instead of 1-3).

```
88
89              #In case user did not select a valid choice
90              else:
91                  print("Please only choose option 1, 2, 3, or 4")
92
93          print("Program Ended") #After loop exits, display this
94
```

Run   🐍 Assignment05 ✕

```
---- 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: 5
Please only choose option 1, 2, 3, or 4
```

Figure 12: Menu choice other than 1-4

At this point, the basic functionality of the program is correct.

Error handling

The final step is incorporating error handling as described in the assignment requirements. There are four structured error handling cases required: File loading, first name user input, last name user input, and file saving.

Error handling requirement 1: File load
The first structured error handling is to box in the operation to read the .json file into the list of dictionary rows upon program startup. When completing Lab03, I wrote code to perform a similar operation, which uses a "try [put original operation here]– except [specific case] – except [general case] – finally [do something]" structure. This structure will attempt to load the file, catch a specific FileNotFoundError if present, otherwise catch a general Exception for other error cases, and then lastly close out the file in the case that the file could not load after opening it within the "try" (with the file thus remaining open).

First, to make sure the overall try-finally portion of the error handling worked, I simply added the desired operations (file open, .json load, and file close) into a "try-finally" block without any error catching in the middle. I confirmed this worked by running the code through it in debug mode. This verifies that within a "try" structure, the file still loads (Figure 13).

```
31
32
33     # When the program starts, read the file data into a list of lists (table)
34     # Extract the data from the file
35     try: #Try to open the file
36         file = open(FILE_NAME,"r")
37         students = json.load(file) #if this breaks, the codee will not reach the next line
38         file.close() #use "finally" at end to close out file if this line not reached
39     finally:
40         if file.closed == False: #have to check if the file didn't close
41             file.close() #and then close it out
42
43     # Present and Process the data
●      while True:
45
46         # Present the menu of choices
47         print(MENU)
48         menu_choice = input("What would you like to do: ")
```

ug   🐍 Assignment05 ×

■ ▶ ▌▌ ⌃ ↓ ↓ ↑ ⊘ ⊘ ⋮   Threads & Variables   Console

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
student_data = {dict: 0} {}
student_first_name = {str} "
student_last_name = {str} "
students = {list: 3} [{'CourseName': 'Python 100', 'FirstName': 'Bob', 'LastName': 'Smith'}, {'CourseName
    0 = {dict: 3} {'CourseName': 'Python 100', 'FirstName': 'Bob', 'LastName': 'Smith'}
    1 = {dict: 3} {'CourseName': 'Python 100', 'FirstName': 'Sue', 'LastName': 'Jones'}
    2 = {dict: 3} {'CourseName': 'py1', 'FirstName': 'jason', 'LastName': 'coult'}
    __len__ = {int} 3
  ⚬ Protected Attributes
  ⊞ Special Variables
```
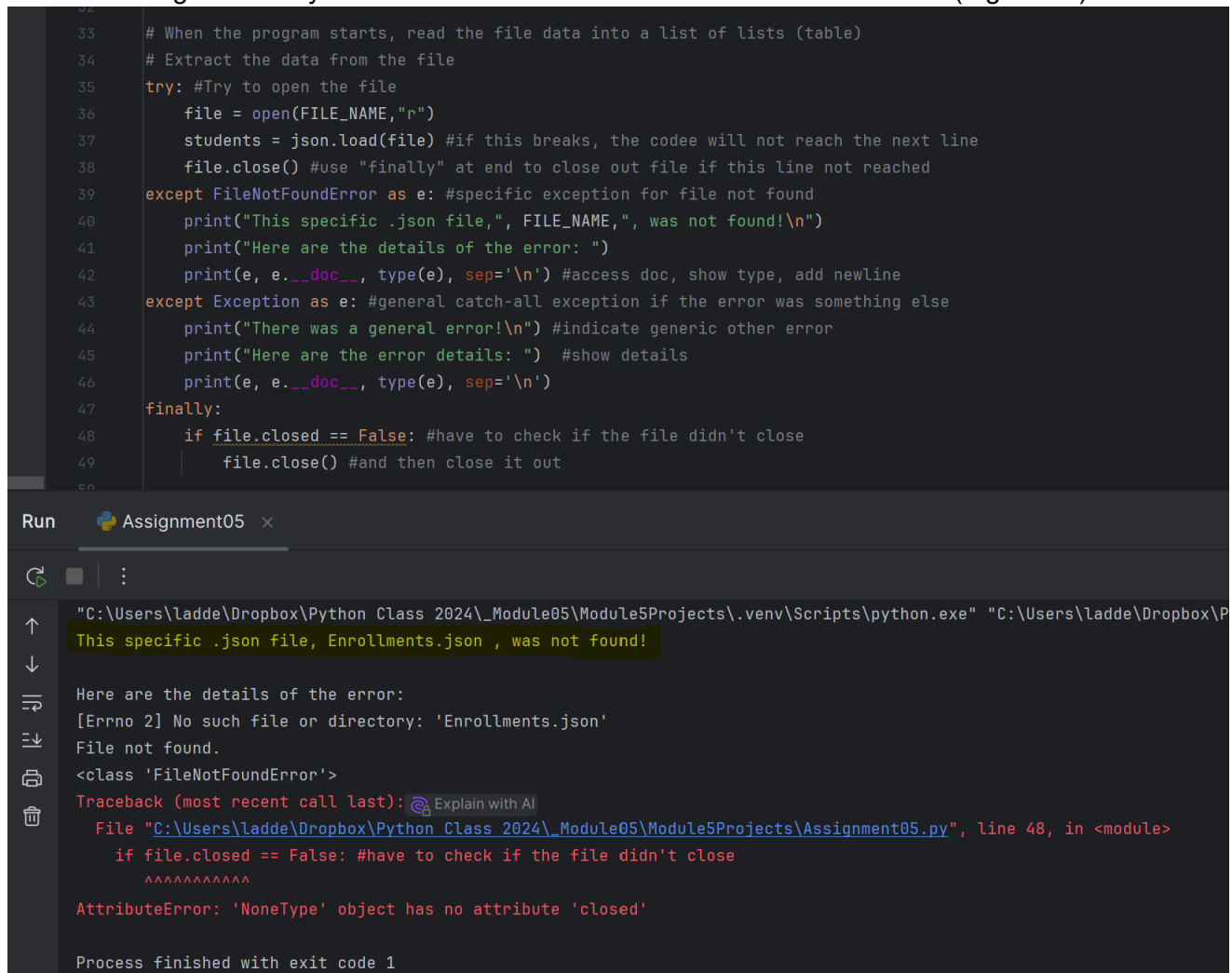
Figure 13: Confirming that file still loads from within "try" structure

Since the basic "try" structure still correctly loads the file, I then completed the structured error handling. I carefully read through the Lab03 code I had written previously, and was able to reuse the same code from my Lab03 project for the specific error handling syntax. I added a string message to state the specific file name that wasn't found (if missing) (Figure 14).

```
# When the program starts, read the file data into a list of lists (table)
# Extract the data from the file
try: #Try to open the file
    file = open(FILE_NAME,"r")
    students = json.load(file) #if this breaks, the codee will not reach the next line
    file.close() #use "finally" at end to close out file if this line not reached
except FileNotFoundError as e: #specific exception for file not found
    print("This specific .json file,", FILE_NAME,", was not found!\n")
    print("Here are the details of the error: ")
    print(e, e.__doc__, type(e), sep='\n') #access doc, show type, add newline
except Exception as e: #general catch-all exception if the error was something else
    print("There was a general error!\n") #indicate generic other error
    print("Here are the error details: ")  #show details
    print(e, e.__doc__, type(e), sep='\n')
finally:
    if file.closed == False: #have to check if the file didn't close
        file.close() #and then close it out
```

Figure 14: Structured error handling for loading .json file into **students** *table*

To check if the error handling worked, I renamed the .json data file (in the assignment folder) to "EnrollmentsZZZ.json" in order to trigger a FileNotFoundError, and then attempted to run the script. The error was caught correctly and indicated the name of the file that wasn't found (Figure 15).


```
33      # When the program starts, read the file data into a list of lists (table)
34      # Extract the data from the file
35      try: #Try to open the file
36          file = open(FILE_NAME,"r")
37          students = json.load(file) #if this breaks, the codee will not reach the next line
38          file.close() #use "finally" at end to close out file if this line not reached
39      except FileNotFoundError as e: #specific exception for file not found
40          print("This specific .json file,", FILE_NAME,", was not found!\n")
41          print("Here are the details of the error: ")
42          print(e, e.__doc__, type(e), sep='\n') #access doc, show type, add newline
43      except Exception as e: #general catch-all exception if the error was something else
44          print("There was a general error!\n") #indicate generic other error
45          print("Here are the error details: ")  #show details
46          print(e, e.__doc__, type(e), sep='\n')
47      finally:
48          if file.closed == False: #have to check if the file didn't close
49              file.close() #and then close it out
```

Run   🐍 Assignment05 ✕

```
"C:\Users\ladde\Dropbox\Python Class 2024\_Module05\Module5Projects\.venv\Scripts\python.exe" "C:\Users\ladde\Dropbox\P
This specific .json file, Enrollments.json , was not found!

Here are the details of the error:
[Errno 2] No such file or directory: 'Enrollments.json'
File not found.
<class 'FileNotFoundError'>
Traceback (most recent call last): 🔍 Explain with AI
  File "C:\Users\ladde\Dropbox\Python Class 2024\_Module05\Module5Projects\Assignment05.py", line 48, in <module>
    if file.closed == False: #have to check if the file didn't close
       ^^^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'closed'

Process finished with exit code 1
```

Figure 15: Catching error when trying to load .json file into **students table**

I then changed the .json file back to its original name and confirmed the code ran without an error.

Error handling requirements 2 and 3: Student's first name and last name
The second and third error cases in need of structured handling are when the student's first and last names are entered. Again as with the file load error handling, I referred to my code from Lab03 which uses "if – raise" blocks to display specific messages to the user when an erroneous input is received.

This assignment requests simply "error handling" but does not specify what types of errors to specifically handle. I decided that when entering a name, one error would be a typo in which a number is entered. Another error would be a user failing to enter any name and leaving an empty string. Both of these errors would be considered "ValueError" types.

To being, I added a basic try – except block around the code for inputting the student's first and last name. To make sure the general structure was functional, I only included an error for the general exception case and ran the code using the debugger to just past the try-except block (Figure 16).

```
57
58          # Input user data
59          if menu_choice == "1":  # This will not work if it is an integer!
60
61              #Enclose in try-except to catch user input errors on first and last names
62              try: #Obtain user input within this try block
63                  student_first_name = input("Enter the student's first name: ") #individual data entry
64                  student_last_name = input("Enter the student's last name: ")
65              except Exception as e: #Catch-all for nonspecific error
66                  print("There was an unknown error when obtaining student name.\n") #Information about error
67                  print("Technical details of error: ") #Show details
68                  print(e, e.__doc__, type(e), sep='\n')
69
●                   course_name = input("Please enter the name of the course: ")
71              student_data = {"CourseName": course_name, #add dictionary entry for single student
72                              "FirstName": student_first_name,
73                              "LastName": student_last_name}
74              students.append(student_data)
75              print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
76              continue
77
```

Assignment05 ✕

Threads & Variables   Console

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

> student_data = {dict: 0} {}
  student_first_name = {str} 'Jas'
  student_last_name = {str} 'Co'

Figure 16: Ensuring a basic try-except structure functions correctly and user input is still collected

After confirming that the general try-except worked, I added specific checks for ValueError cases, using the *raise* function (as in Lab03) within if-blocks to check for specific types of errors. I first added a check for if the field had been left blank (length of the string == 0), and a message indicating the field was empty. I confirmed this worked to check the student's first name for an empty entry (Figure 17).

```
62          try: #Obtain user input within this try block
63
64              #Enter first name and check value for error
65              student_first_name = input("Enter the student's first name: ") #individual data entry
66              if len(student_first_name) == 0:
67                  raise ValueError("The student's first name should not be empty.")
68              elif not student_first_name.isalpha():
69                  raise ValueError("The student's first name should not contain a number.")
70
71              # Enter last name and check value for error
72              student_last_name = input("Enter the student's last name: ")
73
74
75          except ValueError as e: #Catching a specific value-type error
76              print("-" * 50)
77              print("There was a known error in the value of the student's name.") #Specific message about error
78              print(e) #Print error-specific message
79              print("Error details: ")
80              print(e.__doc__,type(e), sep='\n')
81              print("-" * 50)
82
83          except Exception as e: #Catch-all for nonspecific error
84              print("-" * 50)
85              print("There was an unknown error when obtaining student name.") #Information about error
86              print("Technical details of error: ") #Show details
87              print(e, e.__doc__, type(e), sep='\n')
88              print("-" * 50)
89
90          course_name = input("Please enter the name of the course: ")
91          student_data = {"CourseName": course_name, #add dictionary entry for single student
92                          "FirstName": student_first_name,
93                          "LastName": student_last_name}
94          students.append(student_data)
95          print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
96          continue
97
```

Assignment05 ×

```
What would you like to do: 1
Enter the student's first name:
-------------------------------------------------
There was a known error in the value of the student's name.
The student's first name should not be empty.
Error details:
Inappropriate argument value (of correct type).
<class 'ValueError'>
-------------------------------------------------
Please enter the name of the course:
```

Figure 17: Adding specific error handling for empty name input on first name

In the example above in Figure 17, the error is identified correctly, but then the program still proceeds to request the course name rather than exiting menu option 1 even after the erroneous user input (see bottom of console output in figure). Since I do not want erroneous user input to be saved, I moved the collection of *course_name* and appending of data to the *students* table up into the try-block, so that if an error is thrown data will not be saved.

Next, I then added a check for if the input was not just letters (.isalpha), in case the user accidentally input a number. Notably, the .isalpha() function would suffice to check for *both* the case of

an empty input and a number input; however, it does not distinguish between the two cases, requiring a separate check for an empty input in order to provide the user with specific error messaging indicating the field was empty versus if it a number. To test that the isalpha() error type worked in conjunction with moving the course name collection and data appending, I tried an empty name and then mixed alphanumeric name in succession, confirming that they functioned correctly.


```
try: #Obtain user input within this try block

    #Enter first name and check value for error
    student_first_name = input("Enter the student's first name: ") #individual data entry
    if len(student_first_name) == 0:
        raise ValueError("The student's first name should not be empty.")
    elif not student_first_name.isalpha():
        raise ValueError("The student's first name should not contain a number.")

    # Enter last name and check value for error
    student_last_name = input("Enter the student's last name: ")
    course_name = input("Please enter the name of the course: ")
    student_data = {"CourseName": course_name,  # add dictionary entry for single student
                    "FirstName": student_first_name,
                    "LastName": student_last_name}
    students.append(student_data)
    print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
    continue

except ValueError as e: #Catching a specific value-type error
    print("-" * 50)
    print("There was a known error in the value of the student's name.") #Specific message about error
    print(e) #Print error-specific message
    print("Error details: ")
    print(e.__doc__,type(e), sep='\n')
    print("-" * 50)

except Exception as e: #Catch-all for nonspecific error
    print("-" * 50)
    print("There was an unknown error when obtaining student name.") #Information about error
    print("Technical details of error: ") #Show details
    print(e, e.__doc__, type(e), sep='\n')
    print("-" * 50)
```

```
"C:\Users\ladde\Dropbox\Python Class 2024\_Module05\Module5Proj

---- 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: 1
Enter the student's first name:
-----------------------------------------------
There was a known error in the value of the student's name.
The student's first name should not be empty.
Error details:
Inappropriate argument value (of correct type).
<class 'ValueError'>
-----------------------------------------------

---- 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: 1
Enter the student's first name: jason1234
-----------------------------------------------
There was a known error in the value of the student's name.
The student's first name should not contain a number.
Error details:
Inappropriate argument value (of correct type).
<class 'ValueError'>
-----------------------------------------------
```

Figure 18: Code to check for empty first name and numeric first name (left), and console output confirming specific error messages (right)

After confirming that the first name input correctly handles specific value errors, I proceeded to add the same error handling for the last name (Figure 19).

```
if menu_choice == "1":  # This will not work if it is an integer!

    #Enclose in try-except to catch user input errors on first and last names
    try: #Obtain user input within this try block

        #Enter first name and check value for error
        student_first_name = input("Enter the student's first name: ") #individual data entry
        if len(student_first_name) == 0: #check for an empty entry first
            raise ValueError("The student's first name should not be empty.")
        elif not student_first_name.isalpha(): #then check if it contains numbers
            raise ValueError("The student's first name should not contain a number.")

        # Enter last name and check value for error
        student_last_name = input("Enter the student's last name: ")
        if len(student_last_name) == 0: #check for an empty entry first
            raise ValueError("The student's last name should not be empty.")
        elif not student_last_name.isalpha(): #then check if it contains numbers
            raise ValueError("The student's last name should not contain a number.")

        course_name = input("Please enter the name of the course: ")
        student_data = {"CourseName": course_name,  # add dictionary entry for single student
                        "FirstName": student_first_name,
                        "LastName": student_last_name}
        students.append(student_data)
        print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
        continue

    except ValueError as e: #Catching a specific value-type error
        print("-" * 50)
        print("There was a known error in the value of the student's name.") #Specific message about error
        print(e) #Print error-specific message
        print("Error details: ")
        print(e.__doc__,type(e), sep='\n')
        print("-" * 50)

    except Exception as e: #Catch-all for nonspecific error
        print("-" * 50)
        print("There was an unknown error when obtaining student name.") #Information about error
        print("Technical details of error: ") #Show details
        print(e, e.__doc__, type(e), sep='\n')
        print("-" * 50)
```

Figure 19: Menu option 1 code, including error handling for first and last name

I confirmed that the error handling for the last name functioned correctly in the empty case and the numeric case before proceeding.

Error handling requirement 4: Writing dictionary rows to file

The last error handling requirement is for handling the writing of data to a file (using the json.dump() function). To handle this error, I first enclosed the entirety of menu option 4 within a try block that contained only a general error exception and then checked that the operation still worked. I also included a "finally" statement to close the file out in case of an error stopping the program after opening the file (Figure 20).

```
115        # Save the data to a file
116        elif menu_choice == "3":
117
118            try:
119                file = open(FILE_NAME, "w") #open file in write mode
120                json.dump(students, file) #write the students variable to the .json file
121                file.close() #close out file
122
123                #Now display the data, if the file operation worked
124                print("-" * 50)
125                print("Data saved successfully! This is the data saved: ")
126                for student in students:
127                    # display each component of the individual row using the key names
128                    print(f"Student: {student["FirstName"]} {student["LastName"]}, Course: {student["CourseName"]}")
129                print("-" * 50)
130                continue
131
132            except Exception as e: #if it didnt work for another catch-all reason
133                print("A general error has occurred when writing data to file.")
134                print("General error information below: ")
135                print(e, e.__doc__, type(e), sep="\n")
136
137            finally: #then, if there was an error in first block and the file close never was reached
138                if file.closed == False: #check for this
139                    file.close() #then close the file if it wasn't closed due to error in dump
```

🐍 Assignment05 ×

■ ⋮

```
What would you like to do: 1
Enter the student's first name: Kim
Enter the student's last name: Smith
Please enter the name of the course: Class05
You have registered Kim Smith for Class05.

---- 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.
--------------------------------------
```

Enrollments.json × +

File   Edit   View

```
[{"FirstName": "Bob", "LastName": "Smith", "CourseName": "Python 100"},
{"FirstName": "Sue", "LastName": "Jones", "CourseName": "Python 100"},
{"CourseName": "py1", "FirstName": "jason", "LastName": "coult"},
{"CourseName": "Class05", "FirstName": "Kim", "LastName": "Smith"}]
```

Ln 1, Col 1   276 characters   100%   Windows (CRLF)   UTF-8

Figure 20: Confirming that a file still saves when enclosed in try-except-finally block

I then added a TypeError exception to the error handling list as well. To confirm that the try-block would catch an error when writing the file, I attempted to write a file to a dummy file name containing an integer, and observed that indeed the error was caught.

```
115        # Save the data to a file
116    💡 elif menu_choice == "3":
117
118        try:
119            file = open(FILE_NAME, "w") #open file in write mode
120            json.dump(students, dummy) #write the students variable to the .json file
121            file.close() #close out file
122
123            #Now display the data, if the file operation worked
124            print("-" * 50)
125            print("Data saved successfully! This is the data saved: ")
```

🐍 Assignment05 ×

```
4. Exit the program.
----------------------------------------

What would you like to do: 3
------------------------------------------------
A general error has occurred when writing data to file.
General error information below:
'int' object has no attribute 'write'
Attribute not found.
<class 'AttributeError'>
------------------------------------------------


---- Course Registration Program ----
  Select from the following menu:
    1. Register a Student for a Course.
```

Figure 21: Confirming that file write error is handled

At this point, the entire program functions correctly. I confirmed correct console operation within windows command shell by adding a new student with a wrong name, adding a student with a correct name, and saving the file from within the command prompt (Figure 22,).

Figure 22: Confirming program operation and error handling within windows command shell

Figure 23: Confirming .json file contents after adding a new student from within command shell

Again, one errata is that the order of the fields within the dictionary changed in between using the Lab03 .json file and the .json file for this assignment (Figure 23). However, a useful feature of this file type is that the order does not matter, since the fields are accessed within code using key names.

Source control

The assignment requires that the script file and this document be included in a GitHub repository. The repository is located here: https://github.com/jcoult/PythonCourseAssignment05 .

Summary

This assignment demonstrates the utility of Python for loading, manipulating, and saving data formatted in a key-value dictionary format. The JavaScript Object Notation (.json) file format is a commonly used and flexible medium for storing and transferring data formatted in this way. Using .json in Python is relatively simple, in part due to the availability of a library of .json functions that can be imported into Python. This assignment also introduces the use of custom error handling, where error-prone code can be boxed into try-except structures to prevent unforeseen errors from crashing a script and to provide specific error handling depending on the characteristics of the error.