Adam Savage

November 13, 2024

IT FDN 110 A

Assignment 05

https://github.com/adsav54/2024q4_uwp/tree/main

Advanced Collections and Error Handling

Assignment05

Introduction

In Module05 we learned about Dictionaries and using them with files, JSON file format, error handling, and GitHub. These tools align to standard practices and deepen our ability to handle data and collaborate with others. In Assignment05, we modified existing script to use dictionaries, JSON files, and error catching.

Dictionaries

Dictionaries are a Python data structure with similarities to Lists, indicated by curly braces. The main difference is that in lists data is identified by index position while in Dictionaries data is identified by the associated Key, with no positional information as part of the format. Dictionaries are organized by Key:Value pairs, with the position of a Key:Value pair in an array of data having no inherent meaning. Key:Value pairs make Dictionaries 'explicit', in that the identity of a data element is linked to the data and human readable, if desired. In concept, a Dictionary is akin to a single row of a spreadsheet, with Keys akin to column titles and Values as the cell contents. This allows for a collection of Dictionaries to be assembled together into a List, essentially creating a table. Dictionaries can be written to a csv by extracting the elements in a way similar to using List index position, except rather than encoding index position between the brackets the Key is supplied, and formatted into a comma-separated string for writing to file (***Figure 1***).

```
file = open(FILE_NAME, "w")
for student in students:
    file.write(f'{student["FirstName"]},{student["LastName"]},{student["GPA"]}\n')
file.close()
```

Figure 1: Writing a Dictionary-referenced formatted string to file.

Data is read from a csv file into Dictionary format by reading a line from the file, parsing by its delimiter, and mapping the parsed values to a Dictionary by assigning each value to a Key. When looped to handle each row in this way, a List of Dictionaries can be created (***Figure 2***).

```
file = open(FILE_NAME, "r")
for row in file.readlines():
    # Transform the data from the file
    student_data = row.split(',')
    student_data = {"FirstName": student_data[0],
                    "LastName": student_data[1],
                    "GPA": float(student_data[2].strip())}
    # Load it into our collection (list of lists)
    students.append(student_data)
file.close()
```

Figure 2: Reading data from a csv into a Dictionary.

JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is a file format created for javascript that has become widely used because of it's simplicity and readability. In essence, JSON is a format to encode Dictionary-like and List of Dictionaries-like data in a file. It is a series of Dictionaries separated by commas and wrapped in square brackets. Not only is JSON explicitly human-readable but JSON input/output is made very easy using the *json* package. The *json* package is loaded into the script with "import json". Subsequently, JSON files can be read from using 'json.load' and written to using 'json.dump'. The *json* package handles input and output formatting of Dictionaries from and to the JSON file, eliminating the need for assembling Dictionary or List data into a formatted string for writing to file. The *json* package can also interconvert strings and objects using 'loads' and 'dumps'.

Structured Error Handling

Structured error handling is the process which script errors can be captured to provide error information to the user and to execute alternative code. It is important for debugging and for usability, so that expected and unexpected errors don't make the program inoperable. Structured error handling occurs through a series of linked commands governing indented blocks: *try*, *except*, *else*, and *finally*. *try* is the code for which the error handling will applied; *except* determines what happens in response to specific or general errors; *else* is code that runs if no error arises; *finally* is code that runs regardless of whether an error occurred. The last is important for executing code in the context of an error that forces an exit. *try-except* blocks can also be used as a form of data validation, such that unexpected user input can be handled without forcing a stop to the script (***Figure 3***).

```
student_first_name = input("Enter the student's first name: ")
try:
    if not student_first_name.isalpha():
        raise ValueError  # define the exception
except ValueError:
    print("\n=== Please include only alphabetic characters in name. ===")
    continue  # if the user entered non-alphabetic characters, reload the menu
student_last_name = input("Enter the student's last name: ")
try:
    if not student_last_name.isalpha():
        raise ValueError  # define the exception
except ValueError:
```

```
    print("\n=== Please include only alphabetic characters in name. ===")
    continue  # if the user entered non-alphabetic characters, reload the menu
```
Figure 3: Structured error handling invoked as data validation.

GitHub

GitHub is a cloud-based mechanisms for code sharing and collaboration, owned by Microsoft. GitHub is platform agnostic, version controlled, enables complex code development structures, enables team-based real-time collaboration, and enables granular access control. GitHub can mirror one's local environment and allows users to clone a code repository to their local environment. PyCharm can connect to a GitHub repository such that the two are mirrored. This enables one to work in their local environment and seemlessly share they work to others via GitHub.

The Assignment05 script

In Assignment05, we adapted a List- and csv-based script to using Dictionaries and JSON file format. This was useful to simplify the code and for data handling, as the *json* package makes writing arrayed data simple. Rather than needing multiple lines of code and string formatting to correctly write data from a file into an object, a single 'json.load' command does all that behind the scenes (***Figure 4***).

```
# Extract the data from the file – OLD CODE
# file = open(FILE_NAME, "r")
# for row in file.readlines():
#     # Transform the data from the file
#     student_data = row.split(',')
#     student_data = {"FirstName": student_data[0],
#                     "LastName": student_data[1],
#                     "GPA": float(student_data[2].strip())}
#     # Load it into our collection (list of lists)
#     students.append(student_data)
# file.close()

# Extract the data from the file – NEW CODE
file = open(FILE_NAME, "r")
students = json.load(file)
file.close()
```
Figure 4: The *json* package makes file input easy.

In this assignment we also utilized error catching for program robustness and data validation. A modification to the assignment requirements was applied with regards to JSON data input. In the event of a missing file on 'json.load', a *try-except* block creates one, rather than just ending the program with an error (***Figure 5***). This change means a user can get started without having to create a file and reload the program.

```
# Extract the data from the file with error catching for a missing file
try:
    file = open(FILE_NAME, "r")
    students = json.load(file)  # read the JSON into the list of dictionaries
    file.close()
```

```
except FileNotFoundError as e:  # this exception block will create an empty file if none exists
    print("No file exists. Creating an empty file...\n")
    file = open(FILE_NAME, "w")  # opening a file that doesn't exist in write mode creates it
    file.close()
    file = open(FILE_NAME, "r")
except Exception as e:
    print("-- Technical Error Message -- ")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
```

Figure 5: Creating a file in place of missing file using *try-except* error catching.

Summary

In Module05, we took big steps forward to make our code more sophisticated and robust. The use of Dictionaries, JSON files, and error catching brings us closer to a fully functioning console program. Starting our use of GitHub advances the ‘how’ of our scripting, so we can track and share our code.