KaeLi Deng
11/07/2025
Foundations of Programming: Python
Assignment 05
https://rb.gy/ibnwt9

Advanced Data Collections and Error Handling

Introduction

This week we focused on dictionary collections, working with JSON files and basic error handling. I explored how dictionaries store data in key-value pairs and make it easy to organize and access information efficiently. I also practiced reading and writing JSON files, which are widely used for data exchange and storage. In addition, I learned how to use error handling to make programs more stable by catching potential issues and preventing unexpected crashes. Overall, these topics helped me understand how to manage and protect data more effectively in real-world programming tasks.

Dictionary Collections

Unlike a list, a dictionary is a collection that uses labels called keys for each value. This allows data to be accessed quickly using keys instead of index numbers. However, the key names are case-sensitive, which means I need to be careful with uppercase and lowercase letters and make sure they match exactly.

When collecting data into an existing dictionary, I need to open a file (it's better to have some default data in it when it's first opened). Then I can start entering values for each variable. Each key name should be enclosed in double quotes, followed by a colon and the carriable that stores the information, all inside curly brackets {}. Key-value pairs are separated by commas (see Figure 1). Lab 01 provided a clear way to show how dictionary table looks like. This process helps keep the data organized and easy to access.

```
student_first_name = input("What is the student's first name? ")
student_last_name = input("What is the student's last name? ")
student_gpa = float(input("What is the student's GPA? "))

student_data = {"FirstName": student_first_name,
                "LastName": student_last_name,
                "GPA": student_gpa}
students.append(student_data)
continue
```

Figure 1. An example of adding more data to dictionary table from Module 05-Lab 01.

Working with JSON files

JavaScript Object Notation, or JSON, file is a common format for storing and exchanging data. It allows users to work with hierarchical or structured data that CSV format cannot handle. JSON is also human-readable because it uses key-value pairs, making it easy to understand what each value represents by looking at its key name. it is often used to store program settings, user data, and collected information, as well as to share data between different programs or websites.

To save data into a JSON file, I need to import the json module first. I can use json.load() to read data from a file and json.dump() to write data back to it. It's important to use the correct file mode when opening a file --- "r" for reading and "w" for writing. When reading, I use "r" mode in the open() function and call json.load() to load the data into Python. Conversely, I use "w" mode with json.dump() to save the data. Figures 2 and 3 show examples of how the json module is used.

```
#Extract the data form the json file
file = open(FILE_NAME, 'r')
data = json.load(file)
file.close()
```

Figure 2. Example of using json.load() function.

```
file = open(FILE_NAME, 'w')
json.dump(students, file,indent=2)
file.close()
continue
```

Figure 3. Example of using json.dump() function.

Structured Error Handling

Can you anticipate how your users will input data into your program without making mistakes? Structured error handling provides messages that tell users what errors they made while entering data. These errors can include general exceptions, file not found errors, value type errors, zero division errors, custom errors, and so on. Using error handling makes the program more stable, preventing it from crashing due to small mistakes, while also providing clear error messages that help users debug. Additionally, it ensures that resources are released properly.

The main structure of error handling in a program consists of try, which contains the code that might raise an error; except, which executes when a specific error occurs; and finally, which always executes regardless of whether an error occurred, usually used to close files and prevent the program from running indefinitely due to an error.

Figure 4 shows an example of the concepts mentioned above. In this example, e, e.__doc__, type(e), and sep='\n' are also used. Here, e represents the captured exception object. The __doc__ attribute provides a short description of the exception. Type(e) indicates the type of the exception object. Finally, sep='\n' specifies the separator between printed items, and using \n means each item will be printed on a new line.

```
import json

try:
    file = open("Testtesttest.json", 'r')
    data = json.load(file)
except FileNotFoundError as e:
    print("File not found.")
    print(e,e.__doc__, type(e), sep='\n')
except Exception as e:
    print("Something went wrong.")
    print(e,e.__doc__, type(e), sep='\n')
finally:
    file.close()
```

Figure 4. Example of Structured Error Handling

Summary

This week's lessons on dictionary collections, JSON files, and structured error handling have helped me better understand how to manage and protect data in Python. I learned how to organize information efficiently using dictionaries, store and exchange data using JSON, and make programs more stable and user-friendly through error handling. Overall, these skills improve the reliability and readability of my programs and provide a solid foundation for handling more complex data tasks in the future.