Introduction to Programing with Python

Module 05 - Advanced Collections and Error Handling

# Getting Started

Welcome to module 5! In this module, we delve into the fundamental concepts of data structures and file handling in Python. We'll demystify the distinctions between Lists and Dictionaries, clarifying the roles of Indexes and Keys within each. You'll learn the art of writing data to a file from a Dictionary and the reverse process of reading data from a file into a Dictionary, all while unraveling the mysteries of JavaScript Object Notation (JSON) files and how Python's json module can simplify your data handling. We'll also explore the importance of Structured Error Handling and why the Try-Except construct is the recommended approach. Finally, we'll venture into the realm of code storage and sharing, discussing two common locations for your code files and the significance of GitHub in collaborative software development. By the end of this chapter, you'll have a solid foundation in these essential Python concepts and practices.

In the last module, you answered the following questions:

* What is a collection of data?
* What is another word for a collection in Python?
* A String is a collection of what?
* How do you access individual values in a String?
* A Tuple is a collection of what?
* How do you access individual values in a Tuple?
* A List is a collection of what?
* How do you access individual values in a List?
* What functions allow you to read data from a file?
* How do you write data from a String into a text file?
* How do you write data from a List into a text file?
* How do you read data from a text file into a List?

**Exercise**: Take a minute or less to write down a one or two sentence answer to the following question as if you were answering a question from a coworker or interviewer: **What questions do I still have about module 4's content?**

[Getting Started 1](#_Toc155077890)

[Dictionary Collections 2](#_Toc155077891)

[Adding and Removing Data 4](#_Toc155077892)

[Using Dictionaries with File Data 6](#_Toc155077893)

[Demo/Video: Demo01 - Using Dictionaries with files 9](#_Toc155077894)

[Mod05-Lab01: Working with Dictionaries and Files 9](#_Toc155077895)

[Demo/Video: Mod05-Lab01-Review 13](#_Toc155077896)

[JSON Files 13](#_Toc155077897)

[JSON vs CSV 14](#_Toc155077898)

[Working with JSON Files 15](#_Toc155077899)

[Demo/Video: Demo02 - Using JSON Files 17](#_Toc155077900)

[Mod05-Lab02: Working with JSON Files 17](#_Toc155077901)

[Demo/Video: Mod05-Lab02-Review 19](#_Toc155077902)

[Structured Error Handling (Try-Except) 19](#_Toc155077903)

[Avoiding Errors 23](#_Toc155077904)

[Demo/Video: Demo03 - Try-Except 24](#_Toc155077905)

[Mod05-Lab03: Working With Exceptions 24](#_Toc155077906)

[Demo/Video: Mod05-Lab03-Review 26](#_Toc155077907)

[Managing Code Files 27](#_Toc155077908)

[Network File Sharing 27](#_Toc155077909)

[Cloud File Sharing 28](#_Toc155077910)

[GitHub 29](#_Toc155077911)

[Using GitHub's Website 29](#_Toc155077912)

[Demo/Video: Demo04 - Using The GitHub Website 31](#_Toc155077913)

[Mod05-Lab04: Saving Files to GitHub 32](#_Toc155077914)

[Summary 34](#_Toc155077915)

# Dictionary Collections

In the last module you learned how to create a "Table" of data using a list of lists. In this module, you will learn how to create that same table using a list of dictionaries. Here is an example of a two dictionary "rows" being added to a list "Table."

row1 = {"ID":1,"Name":"Bob Smith", "Email":"BSmith@Notmail.com"}  
row2 = {"ID":2,"Name":"Sue Jones", "Email":"SueJ@Yayoo.com"}  
table = [row1, row2]

It can be helpful to think of a dictionary as a row of data in a spreadsheet, where each value in the row can be identified by its column name. However, in a Dictionary these column names are called a "Key."

A screenshot of a computer

Description automatically generated

***Important:*** *Here are some important tips that will help you work with dictionaries.*

* *The* ***Keys in a dictionary are case-sensitive****, so "ID" is not the same as "id" or "Id". You must be careful to always match the key's name exactly or you will get errors!*
* *While our variables names use snake\_casing, we will* ***use TitleCasing name for our keys****. This way we can use a first\_name variable to hold the value from a my\_dict\_data["FirstName"] dictionary items.*
* *Use* ***double-quotes around the key's name****. While this does not matter to Python, it is a requiremet of working with something called JSON which is realated to dictionarys, as we will see shortly.*

Like the Python List class, the Dictionary class has special built-in methods that help you work with the data and keys. Ones that that you use often are items(), values(), and keys(). Here is an example:

*# Create data "rows"*row1: dict = {"ID": 1, "Name": "Bob Smith", "Email": "BSmith@Notmail.com"}  
row2: dict = {"ID": 2, "Name": "Sue Jones", "Email": "SueJ@YaYoo.com"}  
*# Create a data "table"*table = [row1, row2]  
  
*# Display the data in the table*print("\n--- show items in the list 'Table'")  
print(table)  
for row in table:  
 print(row)  
  
*# Display the data using a custom format*print("\n--- Unpacking the elements with the items() function")  
for myKey, myValue in row1.items():  
 print(myKey, " = ", myValue)  
  
*# Display the data jus the values*print("\n--- Displaying only the values()")  
print(row1.values())  
  
*# Display the data jus the keys*print("\n--- Displaying only the keys()")  
print(row1.keys())

A screenshot of a computer

Description automatically generated

OK, there is a lot of code here, so let's break it down.

We are creating a data structure using dictionaries and a list, so first define two dictionaries, row1 and row2, to represent rows of data. Each dictionary contains information about an individual, such as their ID, name, and email. We then create a list named table, which contains these dictionaries.

Next, we print a message to indicate that we are displaying the items in the table. We use a for loop to iterate through each element (row) in the list table. Inside the loop, we print each dictionary representing a row of data.

print(table)  
for row in table:  
 print(row)

After that, we show how to use the items() method of a dictionary (row1) to iterate through its key-value pairs. The for loop unpacks each pair into myKey (the key) and myValue (the value) and then prints them.

for myKey, myValue in row1.items():  
 print(myKey, " = ", myValue)

Then, we use the values() method to print only the values stored in row1.

print(row1.values())

And finally, we use the keys() method to print only the keys stored in row1.

print(row1.keys())

In summary, this code demonstrates how to create and work with dictionaries to store structured data, how to iterate through and access the elements within them, and how to retrieve keys and values from dictionaries. It also shows how to work with lists containing dictionaries to represent tabular data.

**Note:** From here on we will use a list of dictionary rows to store and manage tables of data.

## Adding and Removing Data

Now that we have seen how to create and read data from a dictionary, let's look at how to add and remove data from a list of dictionary rows. Think of this as adding and removing rows of data from a table in a spreadsheet or database.

To add a new row of data you get the data from your user and append that data to a table.

*# Create data "rows"*row1: dict = {"ID": 1, "Name": "Bob Smith", "Email": "BSmith@Notmail.com"}  
row2: dict = {"ID": 2, "Name": "Sue Jones", "Email": "SueJ@YaYoo.com"}  
*# Create a data "table"*table = [row1, row2]

*# Get User Input*print("\n--- Adding more data")

id = input("Enter an ID: ")  
name = input("Enter an Name: ")  
email = input("Enter an Email: ")

row = {"ID": id,"Name": name, "Email": email}

*# Add it to the table*table.append(row)  
  
print("\n--- Displaying the existing and added data")  
for row in table:  
 print(row)

As you can see, the collected values are used to create a new dictionary called row. This dictionary contains three key-value pairs, where the keys are "ID," "Name," and "Email," and the values are the user-provided input values.

Removing data from the table is a little more complicated, but not by much. Here is an example that collects user input for an ID, searches the table for a matching ID, and removes the corresponding entry if found. It then displays the updated table without the removed entry.

*# Get User Input*id = input("Enter an ID to remove: ") *# Remember input returns a string  
  
# Removed data from the table*for row in table:  
 if int(id) == row["ID"]: *# Remember to convert to int!* table.remove(row)  
  
print("\n--- Displaying the existing without the removed data")  
for row in table:  
 print(row)

This code starts by using the input() function to prompt the user to enter an ID to remove. The input value is stored in the variable id. It's important to note that input() returns a string, so the id will initially be a string.

The code then enters a loop that iterates through each dictionary in the table looking for a matching Id. It compares the user-provided id (converted to an integer using int(id)) with the "ID" key in each dictionary in the table (row["ID"]). If a match is found (i.e., the provided id matches the "ID" in a row of data), the table.remove(row) statement is executed. This effectively removes the dictionary (row) from the table list.

After removing the specified entry, the code prints a message indicating that it will display the existing data without the removed entry.

**Important:** Always remember how the comparison works. If you try to match "1" to 1 the comparison is not true. It will only be true if the data is of the same type, integer to integer or string to string. And keep in mind that strings are case sensitive, so if you try matching "Vic" to "vic" it won't work either. A simple way to fix this is to change both to lower case before you compare the strings ( "Vic".lower() == "vic".lower()).

**Exercise**: Take a minute or less to write down a one or two sentence answer to the following question as if you were answering a question from a coworker or interviewer: **What is the difference between and list and a dictionary?**

Top of Form

## Using Dictionaries with File Data

Just like a Python List, you can add user’s input or text file data into a Dictionary. Here is an example of collecting a “row” of data from a user’s input, adding it to a dictionary, then writing the contents of that dictionary to a text file. The newly created row dictionary is appended to the existing list called table. This effectively adds the user's input as a new entry (row) in the data table.

# Data

FILE\_NAME = 'MyData.txt'

file = None

row: dict = {}

table: list = []

# Create the table

row = {"ID": "1","Name":"Bob Smith", "Email":"BSmith@Notmail.com"}

table.append(row)

# Process the data

for objRow in table:

    print(objRow)

# Get User Input

id = input("Enter an ID: ")

name = input("Enter an Name: ")

email = input("Enter an Email: ")

row = {"ID": id,"Name": name, "Email": email}

# Add it to the table

table.append(row)

# Display the contents of each row in the table

for each\_row in table:

    print(each\_row)

    print(each\_row["ID"], each\_row["Name"], each\_row["Email"])

# Write the contents to a file for each row in the table

file = open(FILE\_NAME, 'w')

for each\_row in table:

    file.write(f'{each\_row["ID"]},{each\_row["Name"]},{each\_row["Email"]}\n')

file.close()

A screenshot of a computer

Description automatically generated

This code demonstrates how to create, manipulate, and save tabular data using dictionaries and lists, and how to interact with a file to store this data.

First data is created and processed into a dictionary that represents a row of data with an ID, name, and email. Then it is added the table list using the append() function.

Next, the code collects user input for a new row of data, by prompting the user to enter an ID, name, and email, and then creating a new row dictionary with the user's input. This newly created row is added to the table list, effectively adding the user's input to the table.

Afterward, the code displays the contents of each row in the table, both as a whole dictionary and by accessing individual elements (ID, name, and email).

Finally, the code opens the file specified by FILE\_NAME in write mode ('w'). It then iterates through each row in the table, extracts the ID, name, and email, and writes them to the file as a comma-separated line. Each row is followed by a newline character ('\n'). After writing all the data, the file is closed.

Now let's look at an example of reading a "row" of data from a text file and adding it to a dictionary.

# Data

FILE\_NAME = 'MyData.txt'

file = None

row = {}

table = []

# Process the data

file = open(FILE\_NAME, "r")

for each\_row in file:

    data: list = each\_row.split(",")

    row = {"ID": data[0], "Name": data[1], "Email": data[2].strip()}

    table.append(row)

file.close()

# Display the contents of each row in the table

print(table)  # Note that the default format of a list of dictionaries is not user friendly

# So, let's change it like this

for each\_row in table:

    print(f'{each\_row["ID"]}, {each\_row["Name"]}, {each\_row["Email"]}')

A screen shot of a computer

Description automatically generated

This code reads data from a file, processes it into dictionaries and a list, and then displays the data in a more human-readable format. It demonstrates file reading, data parsing, and list manipulation.

Let's break down and explain this code step by step:

FILE\_NAME = 'MyData.txt'

file = None

row = {}

table = []

In this part, we're setting up some initial data structures and variables, which are quite similar to the previous example.

* **FILE\_NAME** is a string that represents the name of the file we will be working with, 'MyData.txt'.
* **file** is initially set to None and will be used to handle file operations.
* **row** is an empty dictionary that will store data for each row.
* **table** is an empty list that will store multiple rows of data.

Next the data is read and processed from the file by opening the file specified by FILE\_NAME in read mode ('r').

file = open(FILE\_NAME, "r")

for each\_row in file:

    data: list = each\_row.split(",")

    row = {"ID": data[0], "Name": data[1], "Email": data[2].strip()}

    table.append(row)

file.close()

We iterate through each line (each\_row) in the file. For each line, we split it using a comma (,) as the delimiter, which separates the data into a list called "data."

We create a dictionary row with the extracted data, where data[0] represents the ID, data[1] represents the name, and data[2] (with .strip()) represents the email. We then append this row dictionary to the table list. After processing all the lines in the file, we close the file using file.close().

Finally, we display the contents of the table. We first print the table itself, which will display the list of dictionaries containing the data in its default format, like this:

[{'id': '1', 'name': 'Bob Smith', 'email': 'BSmith@Notmail.com'}, {'id': '2', 'name': 'Sue Jones', 'email': 'SJones@Yayou.com'}]

This isn't a format that most users expect, so instead, we iterate through each dictionary in the table and use f-strings to format and print the ID, name, and email from each dictionary, making it more user-friendly.

for each\_row in table:

    print(f'{each\_row["ID"]}, {each\_row["Name"]}, {each\_row["Email"]})

As you can see, writing and reading from a file using a dictionary is not much different than working with a list. However, when working with files, there are some differences to consider when using dictionaries compared to lists.

Here are some key differences to keep in mind when reading and writing data to files using these data structures:

Data Organization:

* List: When using a list to read data from a file, you typically store each line as an element in the list. Each element corresponds to a single line in the file.
* Dictionary: When using a dictionary, you often map specific keys to values for each line in the file. Each key-value pair represents data from a single line, making it easy to access specific fields using keys.

Accessing Data:

* List: To access data read from a file using a list, you use index numbers to retrieve specific lines or elements.
* Dictionary: With dictionaries, you can access specific pieces of data by using the associated keys, making it more intuitive to retrieve specific fields of interest.

Data Structure Choice:

* List: Lists are suitable for scenarios where each line in the file has a consistent structure, and you need to process the data sequentially. For example, reading a CSV file where each row has the same columns.
* Dictionary: Dictionaries are handy when the data in the file has distinct key-value pairs, such as reading configuration files or data with named attributes.

Data Validation:

* List: Lists do not provide inherent validation for the data structure. It's up to the programmer to ensure the correct indexing and data types.
* Dictionary: Dictionaries offer a more structured way to validate data because you can check if specific keys exist and verify their values.

Data Transformation:

* List: Lists may require additional processing to transform the data into a usable format, especially if the data within each line needs to be split or parsed further.
* Dictionary: Dictionaries often simplify data transformation, as you can directly access and manipulate individual values based on keys.

Field Names:

* List: Lists do not inherently store field names. You need to rely on positional indexing to understand the meaning of each element.
* Dictionary: Dictionaries inherently associate field names (keys) with their corresponding values, making it easier to understand the data structure.

In the end, the choice between using lists or dictionaries when reading and writing data from files depends on the structure and nature of the data within the file. Dictionaries are well-suited for structured data with named attributes, while lists are more appropriate when the data is relatively simple and can be processed sequentially.

## Demo/Video: Demo01 - Using Dictionaries with files

**Exercise**: Take a minute or less to write down a one or two sentence answer to the following question as if you were answering a question from a coworker or interviewer: **What is the difference between using a list and a dictionary when working with csv files?**

# Mod05-Lab01: Working with Dictionaries and Files

|  |
| --- |
|  |

In this lab, you will create a Python program that demonstrates how to save student's GPA data to a file from a dictionary. This lab also covers how to interact with the user by taking input and offering choices.

1. **Open** the **PythonLabs** PyCharm project you created in Mod03-Lab01.

2. **Create** a new Python file named "Mod04-Lab01-WorkingWithFileData.py".

3. **Add** the following pseudo-code to the file. **Remember to change** the placeholders in the script header to your name, the current date, and set the activity to "created script".

*# ------------------------------------------------------------------------------------------ #  
# Title: Working With Dictionaries And Files  
# Desc: Shows how work with dictionaries and files when using a table of data  
# Change Log: (Who, When, What)  
# <Your Name>,<The currentdate>,Created Script  
# ------------------------------------------------------------------------------------------ #  
  
# Define the program's data  
  
# Define the program's data  
  
# When the program starts, read the file data into a list of dictionary rows (table)  
# Extract the data from the file  
 # Transform the data from the file  
 # Load it into the collection   
  
# Repeat the follow tasks  
 # display the table's current data  
 # Add data to the table  
 # Save the data to the file  
 # Exit the program*

4.**Define** and **Initialize** the necessary data variables and a constant to store information about the student, message, and file data .

*# Define the Data Constants*FILE\_NAME: str = 'MyLabData.csv'  
  
*# Define the program's data*MENU: str = '''  
---- Student GPAs ------------------------------  
 Select from the following menu:   
 1. Show current student data.   
 2. Enter new student data.  
 3. Save data to a file.  
 4. Exit the program.  
--------------------------------------------------   
'''  
  
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.*student\_gpa: float = 0.0 *# Holds the GPA of a student entered by the user.*message: str = '' *# Holds a custom message string*menu\_choice: str = '' *# Hold the choice made by the user.*student\_data: dict = {} *# one row of student data*students: list = [] *# a table of student data*file\_data: str = '' *# Holds combined string data separated by a comma.*file = None *# Not using type hint helps PyCharm, so we won't use it going forward*

5. **Run** your script and troubleshoot any error that occurs.

**Important:** Now that our code is getting more complex, you should run your code to test each section. These small tests will greatly reduce errors since you can troubleshoot any issue that arises as some as possible, before the code becomes even more complex!

6. **Create** a "for" loop that processes the contents of a file. The processing should split the text data into a list, strip the new-line from the end GPA value, cast it as a floating point value, and add the resulting data to a list of lists.

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()

7. **Create** a file called '**MyLabData.csv**' and add the following starter data:

Bob,Smith,4.0

Sue,Jones,3.8

**Notes:**

* Make sure to name it 'MyLabData.csv' or you will get an error when you test your code
* Remember to add a new line after the last entry or you will get an error when you test your code

8. **Run** your script and **troubleshoot** any error that occurs.

9. **Create** a while loop that continues to run the code until the break command is encountered. Print out a menu of options and ask the user to make a choice each time the loop repeats.

while True:  
  
 print(MENU)  
 menu\_choice = input("Enter your menu choice number: ")  
 print() *# Adding extra space to make it look nicer.*

10. **Run** your script and troubleshoot any error that occurs. Remember, this will result in an endless loop, so you will have to manually stop the script.

11. **Add** code so that if the user chooses option 1, it will evaluate the student's GPA and determine and set a personalized message based on four GPA ranges (1.0, 2.0,3.0,4.0). Include an "else" option when none of the conditions are met, then display the selected message, based on the GPA analysis.

if menu\_choice == "1":  
 *# Process the data to create and display a custom message* print("-"\*50)  
 for student in students:  
 if student["GPA"] >= 4.0:  
 message = " {} {} earned an A with a {:.2f} GPA"  
 elif student["GPA"] >= 3.0:  
 message = " {} {} earned a B with a {:.2f} GPA"  
 elif student["GPA"] >= 2.0:  
 message = " {} {} earned a C with a {:.2f} GPA"  
 elif student["GPA"] >= 1.0:  
 message = " {} {} earned a D with a {:.2f} GPA"  
 else:  
 message = " {} {}'s {:.2f} GPA was not a passing grade"  
  
 print(message.format(student["FirstName"], student["LastName"], student["GPA"]))  
 print("-"\*50)  
 continue

12. **Run** your script and troubleshoot any error that occurs. Remember, this will result in an endless loop, so you will have to manually stop the script.

13. **Add** code so that if the user chooses option 2, prompt the user to input the student's first name, last name, and GPA using the input function. Include the float() function to convert the GPA input.

elif menu\_choice == "2":  
 *# Input the data* 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? "))

14. **Add** the student data to a dictionary using the student\_data variable, then **add** that list to the students list to create a table of data (a dictionary inside of a list).

student\_data = {"FirstName": student\_first\_name,  
 "LastName": student\_last\_name,  
 "GPA": student\_gpa}  
 students.append(student\_data)  
 continue

15. **Run** your script and troubleshoot any error that occurs. Remember, this will result in an endless loop, so you will have to manually stop the script.

16. **Add** code so that if the user chooses 3, will will write the data from the dictionary into the file.

elif menu\_choice == "3":  
 *# Save the data to the file* file = open(FILE\_NAME, "w")  
 for student in students:  
 file.write(f'{student["FirstName"]},{student["LastName"]},{student["GPA"]}\n')  
 file.close()  
 print("Data Saved!")  
 continue

17. **Test** your code by running it and adding the following input, then reviewing the file to verify that the new data is in the file.

Vic,Vu,3.5

**Important:** Now that you know the script runs in PyCharm, it still needs to be tested in OS command/shell. After all, the person using your program should not be expected to install and learn to use PyCharm just to run your program.

18. Open your script file in Windows Explorer or Mac OS Finder.

19. Copy your script file to a folder that will be easy to access, such as in your "**documents/PythonCourse**" folder.

20. **Open** a command shell terminal window.

21. **Navigate** to the location of your script file.

22. **Run** and test the script using the Python interpreter program Python.exe or Python3.exe.

After completing this lab, please watch the following video review:

## Demo/Video: Mod05-Lab01-Review

In this lab, you created an application that continuously prompts the user to input student data, offers the option to display this data when requested, allows the user to save the data, and allows for a graceful program exit by exiting the loop. Now, let's look at an alternative way of storing our data using something a bit more complex than a csv file.

|  |
| --- |
|  |

# JSON Files

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for both humans to read and write and for machines to parse and generate. It's widely used for data serialization and communication between a client and a server, as well as for configuration files and data storage. Here's an explanation of JSON files:

Basic Structure:

* JSON files consist of key-value pairs (very much like Python dictionarys do!)
* Keys are strings enclosed in double quotes.
* Values can be strings, numbers, objects, arrays, booleans, or null.
* Data is organized using curly braces {} for objects and square brackets [] for arrays.
* Commas , separate key-value pairs or elements within an array.

Here is an example of

{

  "Name": "John Doe",

  "Age": 30,

  "City": "New York",

  "IsStudent": false,

  "Hobbies": ["reading", "swimming", "coding"]

}

In this JSON object, there are five key-value pairs, where "Name," "Age," "City," "IsStudent," and "Hobbies" are keys, and their corresponding values are strings, numbers, booleans, and an array.

Data Types:

* Strings: Enclosed in double quotes (e.g., "Name": "John Doe").
* Numbers: Can be integers or floating-point numbers (e.g., "age": 30).
* Objects: Nestable key-value pairs enclosed in curly braces (e.g., "address": {"street": "123 Main St", "city": "Boston"}).
* Arrays: Ordered lists of values enclosed in square brackets (e.g., "Hobbies": ["reading", "swimming", "coding"]).
* Booleans: Represented as true or false (e.g., "IsStudent": false).
* Null: Represented as null (e.g., "spouse": null).

**Note:** JSON allows for nesting objects and arrays within objects and arrays, allowing you to represent complex data structures. This is an advanced concept you should be aware of, but we will not need something this complicated in our course.

Use Cases:

* Data Exchange: JSON is commonly used to exchange data between a client and a server in web applications. It's a format that both front-end and back-end systems can easily work with.
* Configuration Files: Many software applications use JSON for configuration files because of its human-readable format.
* Data Storage: JSON is used for storing structured data, and it's often used in NoSQL databases that store data in document-like structures.
* API Responses: APIs often return data in JSON format, making it easy for applications to consume the data.

## JSON vs CSV

JSON is a flexible and hierarchical format suitable for representing structured data with varying complexity. It's commonly used for configuration, web APIs, and document-oriented databases. On the other hand, CSV is a simpler tabular format primarily used for data storage, import/export, and analysis, especially when dealing with large datasets.

The choice between JSON and CSV depends on the specific requirements and characteristics of the data and the intended use cases. Here are some common differences and use cases between JSON and CSV files:

Data Structure:

* JSON is a hierarchical data format consisting of key-value pairs, where keys are strings enclosed in double quotes, and values can be strings, numbers, objects, arrays, booleans, or null.
* JSON is more flexible in representing complex and nested data structures, making it suitable for data with varying levels of hierarchy.
* CSV is a flat file format that stores data in tabular form, with rows and columns.
* CSV is simpler and flatter than JSON and doesn't support nested structures by default. It primarily represents data as plain text separated by commas (or other delimiters).

Readability:

* JSON is human-readable, with clear key-value pairs and data structures. It is often used for configuration files and settings because of its readability.
* CSV is human-readable to some extent, but it may become less intuitive for complex data with many columns. It is more suited for large datasets with a straightforward tabular structure.

Complexity

* JSON is used for data interchange between systems, especially in web applications where JavaScript is commonly used. It can easily represent data with mixed data types and nested structures, making it suitable for APIs.
* CSV is typically used for exporting and importing data to and from applications like spreadsheets or databases. It's often used for tabular data that needs to be consumed by software for analysis or reporting.

Use Cases:

* JSON is commonly used for configuration files in software applications. It is used for data exchange in web APIs, where data can be more complex and hierarchical.
* JSON is suitable for NoSQL databases that store data in a document-oriented format. JavaScript-based applications often use JSON for data storage and communication.
* CSV is often used for data import and export in spreadsheet software (e.g., Microsoft Excel, Google Sheets). It is favored for storing large datasets in a structured format, making it easier to analyze using data analysis tools.
* CSV is commonly used for database backup and restoration due to its simplicity.

Data Integrity:

* JSON data is less prone to data type issues as it can represent a variety of data types explicitly. JSON may include data validation through schemas like JSON Schema.
* CSV is more vulnerable to data type issues since it treats all values as plain text. It's essential to ensure that data types are consistent during CSV handling.

File Size:

* JSON files tend to be larger than equivalent CSV files because of the additional structure (keys, objects, arrays) and the use of human-readable text. If there is a large about of data, like working with large data imports, that can impact performance and storage.
* CSV files are generally more compact than JSON files because they only contain raw data and minimal delimiters. This is one of the reasons for it being the most common format used when transfering or importing larges amounts of data.

In the end, the choice between using JSON and CSV depends on the specific requirements and characteristics of the data and the intended use cases. In most cases, you will often have to work with whichever format the software, or team you are working with, dictate.

## Working with JSON Files

To write the Python dictionary and list you provided to a JSON file, you can either do it manually by converting it into a string or you can use the "json" module in Python.

Using the string option is very much like the earlier example.

# Define your data

row1 = {"ID": 1, "Name": "Bob Smith", "Email": "BSmith@Notmail.com"}

row2 = {"ID": 2, "Name": "Sue Jones", "Email": "SueJ@YaYoo.com"}

table = [row1, row2]

# Write the contents to a file for each row in the table

file = open("data.json", 'w')

file.write('[\n')

file.write(f'{{"ID": "{row1["ID"]}", "Name": "{row1["Name"]}", "Email": "{row1["Email"]}"}},\n')

file.write(f'{{"ID": "{row2["ID"]}", "Name": "{row2["Name"]}", "Email": "{row2["Email"]}"}}\n')

file.write(']')

file.close()

'''

 The result in the file looks like this:

[

{"ID": "1","Name": "Bob Smith","Email": "BSmith@Notmail.com"},

{"ID": "2","Name": "Sue Jones","Email": "SueJ@YaYoo.com"}

]

'''

**Note:** We are using two curly braces to create a single brace within the f'string. You can use two braces, brackets, or parentheses in this way an escape sequence.

Now, this is not too hard to follow, but as the data gets more complicated it may be best to use the json module. The json module in Python is a built-in module that provides methods for working with JSON (JavaScript Object Notation) data. JSON is a lightweight data interchange format that is easy for humans to read and write, and it's also easy for machines to parse and generate. Here is an example:

import json # import code from Python's json module into my script

import json

# Define your data

row1 = {"ID": 1, "Name": "Bob Smith", "Email": "BSmith@Notmail.com"}

row2 = {"ID": 2, "Name": "Sue Jones", "Email": "SueJ@YaYoo.com"}

table = [row1, row2]

# Write the contents to a file for each row in the table

file = open("data.json", 'w')

json.dump(table, file)

file.close()

'''

 The result in the file are still the same:

 [{"ID": 1, "Name": "Bob Smith", "Email": "BSmith@Notmail.com"},

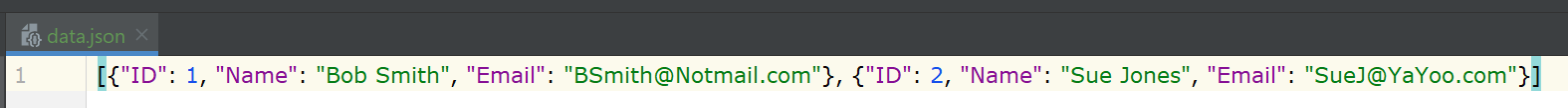
  {"ID": 2, "Name": "Sue Jones", "Email": "SueJ@YaYoo.com"}]

'''

In this code:

* We import the json module, which provides functions for working with JSON data.
* We define your data in the form of dictionaries (row1 and row2) and a list (table) containing these dictionaries.
* We open the file using the open() function with the mode "w" (write mode), which allows us to write data to the file. We use a with statement to ensure that the file is properly closed after writing.
* We use the json.dump() function to write the table list (which contains dictionaries) to the JSON file. This function write the data into the specified file in a JSON format.

After running this code, you will have a JSON file named "data.json" in your working directory containing the data from the table list in JSON format.



To read the data created by the code you provided, which writes a list of dictionaries (table) to a JSON file named "data.json," you can use the following code to read and process that JSON data:

import json

# Open the JSON file for reading

file = open("data.json", "r")

data = json.load(file)

# Now 'data' contains the parsed JSON data as a Python list of dictionaries

for item in data:

    print(f"ID: {item['ID']}, Name: {item['Name']}, Email: {item['Email']}")

This code is very similar to the previous example. It uses the json.load(file) method to parse the JSON data from the "data.json" file into a Python list of dictionaries (data), and then it loops through the list to print the values associated with "ID," "Name," and "Email" for each row.

A screen shot of a computer

Description automatically generated

## Demo/Video: Demo02 - Using JSON Files

**Exercise**: Take a minute or less to write down a one or two sentence answer to the following question as if you were answering a question from a coworker or interviewer: **What is the json module used for in Python?**

# Mod05-Lab02: Working with JSON Files

|  |
| --- |
|  |

In this lab, you will modify the program you made in the last lab to save student's GPA data to a json file. Since this lab is very similar to the last one, we only need to change some of the code.

**Note:** If you could not get the last lab to work, start by using the code in the Mod05-Lab01 answer file.

1. **Open** the **PythonLabs** PyCharm project (if you closed it.)

2. **Save** the Python file "Mod04-Lab01-WorkingWithFileData.py" from lab 1 **as** "**Mod04-Lab02-WorkingWithJSONFiles.py**".

3. **Add** an import statement at the after the header to import code from the json module.

import json

4. **Change** the file name to use the json extension instead of a csv.

*# Define the Data Constants*FILE\_NAME: str = 'MyLabData.json

5. **Locate and comment out** the code that opens and reads the csv data into the table.

*# 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()*

6. **Add** code that opens the json file, dumps the data into the students table variable and closes the file again.

file = open(FILE\_NAME, "r")  
students = json.load(file)  
file.close()

7. **Create** a file called '**MyLabData.json**' and add the following starter data:

[{"FirstName": "Bob", "LastName": "Smith", "GPA": 3.5},

{"FirstName": "Sue", "LastName": "Jones", "GPA": 4.0}]

**Note:** Make sure to name it 'MyLabData.json' or you will get an error when you test your code

8. **Run** your script and **troubleshoot** any error that occurs. Make sure to use the menu option that displays the current data.

9. **Comment out the** code used when the user chooses 3

elif menu\_choice == "3":

*# Save the data to the file  
# file = open(FILE\_NAME, "w")  
# for student in students:  
# file.write(f'{student["FirstName"]},{student["LastName"]},{student["GPA"]}\n')  
# file.close()  
# print("Data Saved!")*

10. **Add** code that will load the student data into the json file:

*# Save the data to the file*file = open(FILE\_NAME, "w")  
json.dump(students, file)  
file.close()  
continue

11. **Test** your code by running it and adding the following input, then reviewing the file to verify that the new data is in the file.

Vic,Vu,3.5

**Important:** Now that you know the script runs in PyCharm, it still needs to be tested in OS command/shell. After all, the person using your program should not be expected to install and learn to use PyCharm just to run your program.

12. Open your script file in Windows Explorer or Mac OS Finder.

13. Copy your script file to a folder that will be easy to access, such as in your "**documents/PythonCourse**" folder.

14. **Open** a command shell terminal window.

15. **Navigate** to the location of your script file.

16. **Run** and test the script using the Python interpreter program Python.exe or Python3.exe.

After completing this lab, please watch the following video review:

## Demo/Video: Mod05-Lab02-Review

In this lab, you modified the application to save the data as a JSON file. Now, let's look at some options that might improve our code by adding error handling.

|  |
| --- |
|  |

# Structured Error Handling (Try-Except)

When you are programming, you fix your bugs immediately and make sure the code runs smoothly. However, it often happens that other people introduce new bugs when they use your program. Error Handing improves your scripts by managing errors you may not have control over in any other way. For example, you may ask a user for input of a number to calculate but receive a character instead. While you cannot stop people from making mistakes you can actively plan for them and make the process of recovering from errors less painful.

Most modern languages allow you to trap errors in your programs using a try-except construct. One advantage is its ability to provide more general and user-friendly error messages. Another advantage is that it allows for a simple, organized way to manage error statements.

A third advantage is that if an error occurs in the grouped statements, Python automatically moves to another set of statements where you can customize handling an error in your own way (instead of the way Python would normally do so).

For example, normally when Python encounters a "divide by zero" error from code like this:

value1: float = 3.0

value2: float = 0.0

print(value1 / value2)

A screenshot of a computer

Description automatically generated

it completely stops the program and outputs a message like this.

Traceback (most recent call last):

  File "C:\\_PythonClass\\_Module05\Demos\Test.py", line 3, in <module>

    print(value1 / value2)

          ~~~~~~~^~~~~~~~

ZeroDivisionError: float division by zero

This technical explanation is intended to help developers fix their code, and with practice it often does. However, while technical messages work fine for developers, they are often too complex for end-users.

Adding a Try-Except block to your code allows you to customize the error messages as shown here.

    value1: float = 3.0

    value2: float = 0.0

    print(value1 / value2)

except:

print("Error! Please check you are not dividing by zero.")

A screenshot of a computer

Description automatically generated

### Using the Exception Class

In this code, we are using a "try-except" block to handle errors gracefully. However, it may now be too generic for more technical users, but we can easily improve this by providing both the simple and complex message.

To do this, we can use the "Exception" class. As we will see soon, classes are a named set of constants, variables, and functions. In this case, the Exception class holds information about the error that just occurred when the code is running.

Python automatically creates an Exception object using the Exception class when an error occurs. The Exception object automatically fills with information about the error that caused the exception.

You can capture the Exception object in the except section of a try-except block and extract the error messages. In this next example, we are using a variable named "e" to capture a reference to the automatically generated Exception object.

try:  
 value1: float = 3.0  
 value2: float = 0.0  
 print(value1 / value2)  
except Exception as e:  
 print("Error! Please check you are not dividing by zero.\n")  
 print("-- Technical Error Message -- ")  
 print(e) *# Print the exception object (typically includes the error message)* print(type(e)) *# Print the type of the exception object* print(e.\_\_doc\_\_) *# Print the documentation string of the exception type* print(e.\_\_str\_\_()) *# Print the string representation of the exception*

This time, the error message is a combination of simple and complex, allowing uses with different skill levels to choose. The "e.\_\_doc\_\_" code is a built-in attribute in Python that represents the documentation string (docstring) associated with an object and will typically provide a brief description of the exception type and what it signifies. The "e.\_\_str\_\_() " code typically returns a string providing details about the exception, including its error message. For example, it might return something like "division by zero" or "file not found."

Error! Please check you are not dividing by zero.

Built-In Pythons error info:

float division by zero

<class 'ZeroDivisionError'>

Second argument to a division or modulo operation was zero.

float division by zero

We said "typically" because different developers created different messages for different exceptions, and they are not always consistent. You often must test the built-in error messages before you can tell if you want to include e.\_\_doc\_\_, e.\_\_str\_\_(), or both. However, including a custom friendly message for non-technical users is almost always appreciated!

**Tip:** The syntax of "Exception as e:" may look weird to you! This syntax is actually from other languages where they put the variables type, in this case the Exception class, before the name of the variable, in this case "e." There are several code statements in Python that can look out of place, often these were added to the Python language by a developer that was used to working in another language. It's best to just accept it as part of Python's unique "personality!"

### Catching Specific Exceptions

The Exception class can catch any type of error, but you can catch specific errors using more specific exception classes. These classes are based on the general Exception class, but each has been modified to be about a specific type of error. Here are some examples:try:  
 f = open('SomeFile.txt', 'r')  
except ZeroDivisionError as e:  
 print("Please do not use Zero for the second number!\n")  
 print("Built-In Python error info: ")  
 print(e, e.\_\_doc\_\_, type(e), sep='\n')  
except FileNotFoundError as e:  
 print("Text file must exist before running this script!]\n")  
 print("Built-In Python error info: ")  
 print(e, e.\_\_doc\_\_, type(e), sep='\n')  
except Exception as e:  
 print("There was a non-specific error!\n")  
 print("Built-In Python error info: ")  
 print(e, e.\_\_doc\_\_, type(e), sep='\n')

A screenshot of a computer error

Description automatically generated

Here's what each part of the code does:

First the code inside the try block attempts to open a file named 'SomeFile.txt' in read mode ('r') using the open() function. Unlike write mode, the read mode only works when the file already exist. We made sure that it did not when test this code so that it would "raise" an exception.

This time, there are three exception blocks that handle different types of exceptions.

* The first except block catches a ZeroDivisionError. However, in this code, there is no division operation, so this block will not be executed, and Python moves on the the next exception block.
* The second except block catches a FileNotFoundError if it occurs. It prints a specific error message for this case and additional information about the error, such as the error message, documentation string, and the type of the exception (FileNotFoundError). This is the exception block that runs!
* Generic Exception: The third except block is a catch-all for any other exceptions that may occur. It provides a more general error message and displays the error message, documentation string, and the type of the exception. In our example these line of code never run, since an earlier exception block caught the error.

***Note:*** *You can find a list of Python's exception classes on this webpage:* <https://docs.python.org/3/library/exceptions.html#bltin-exceptions>

### Raising Custom Errors

Python automatically generates errors based on conditions defined by the Python Runtime. However, you can also "raise" errors based on custom conditions.

try:

    new\_file\_name = input("Enter the name of the file you want to make: ")

    # Check for a number in the zeroth character

    if new\_file\_name[0].isnumeric():

        raise Exception('Please, do not start the file\'s name with a number')

except Exception as e:

    print(e)

A screenshot of a computer

Description automatically generated

The purpose of this code is to validate the user's input for a file name and prevent them from creating a file with a name that starts with a number. If the condition in the if statement is True, meaning the file name starts with a number, the code creates a custom exception object using the raise statement.

The custom exception objects will include the specific error message put there by the raise statement, "Please, do not start the file's name with a number".

Here's a more complex example, using the ValueError exception object to handle a specific type of error: when the user enters a last name that contains characters other than alphabetic characters (A through Z). To accomplish this, we're using the isalpha() function, which returns True only if all the characters in the input string are alphabetic.

try:  
 *# Check that the input does not include numbers* student\_last\_name = input("Enter the student's last name: ")  
 if not student\_last\_name.isalpha():  
 raise ValueError("The last name should not contain numbers.")  
except ValueError as e:  
 print(e) *# Prints the custom message* print("-- Technical Error Message -- ")  
 print(e.\_\_doc\_\_)  
 print(e.\_\_str\_\_())  
except Exception as e:  
 print("There was a non-specific error!\n")  
 print("Built-In Python error info: ")  
 print(e, e.\_\_doc\_\_, type(e), sep='\n')

A screenshot of a computer

Description automatically generated

In this except block, there are two except clauses. The first except ValueError as e: block catches and handles a specific ValueError exception that may be raised if the input contains numbers, printing both the custom error message (e) that was raised with the ValueError and the technical error information, such as the docstring (e.\_\_doc\_\_) and the string representation (e.\_\_str\_\_()) of the exception. The second except Exception as e: block is a catch-all for handling any other exceptions (i.e., exceptions that are not ValueError).

This structure allows you to handle specific exceptions differently while providing general error handling for any other unexpected exceptions. It's good practice to handle exceptions with more specificity when possible.

## Avoiding Errors

The exception block is not just to displaying messages. It can also be used to avoid errors later in your code. For example, if you open a file, then have an exception before it closes, the file will remain open! This may cause another error later, when you try to open the same file that is already open. To avoid this possibility, you can add code to test this condition and close the file as necessary. Here is an example:

try:

    file = open('SomeFile.txt', 'r')

except FileNotFoundError as e:

**if file.closed == False:** # Make sure the file is open before trying to close it.

**file.close()**

    print("Text file must exist before running this script!]\n")

    print("Built-In Python error info: ")

    print(e, e.\_\_doc\_\_, type(e), sep='\n')

except Exception as e:

**if file.closed == False:** # Make sure the file is open before trying to close it.

**file.close()**

    print("There was a non-specific error!\n")

    print("Built-In Python error info: ")

    print(e, e.\_\_doc\_\_, type(e), sep='\n')

The "if file.closed == False:" statement checks whether the file is closed or not by accessing the "closed" attribute of the file object (file). If the file is not closed (i.e., file.closed evaluates to False), it means the file is still open. This code is a defensive measure that checks if a file is open within an except block designed to handle exceptions. If the file is open, it is closed to ensure proper resource management and to prevent any potential issues that might arise from having an unclosed file.

## Demo/Video: Demo03 - Try-Except

**Exercise**: Take a minute or less to write down a one or two sentence answer to the following question as if you were answering a question from a coworker or interviewer: **What is a structured error handling?**

# Mod05-Lab03: Working With Exceptions

|  |
| --- |
|  |

In this lab, you modify the program you made in the last lab to use structured error handling. Since this lab is very similar to the last one, we only need to change some of the code.

**Note:** If you could not get the last lab to work, start by using the code in the Mod05-Lab02 answer file.

1. **Open** the **PythonLabs** PyCharm project (if you closed it.)

2. **Save** the Python file "Mod04-Lab02-WorkingWithFileData.py" from lab 1 **as** "**Mod04-Lab03-WorkingWithExceptions.py**".

3. **Locate** the code that opens and reads the json data into the table.

4 **Add** a **try** statement **before** that code, indent the code, **then add** the following exception handling **after** the code.

try:

file = open(FILE\_NAME, "r")  
 students = json.load(file)  
 file.close()

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')  
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 file.closed == False:  
 file.close()

5. **Run** your script and **troubleshoot** any errors that occur, until the program runs without errors.

6. **Locate and change** the following code to test the FileNotFoundError exception block.

FILE\_NAME: str = 'MyLabData.jsonZZZ'

7. **Run** your script and **verify** that the error displays the **"Text file must exist before running this script!"** message.

8. **Change** the code back to its original value, **Run** your script and **troubleshoot** any errors that occur, until the program runs without errors once again.

9. **Locate** the code that inputs user data into the table.

10. **Add** a **try** statement **before** that code, indent the code, **then add** the following validation and exception handling **after** the code.

try:  
 *# Input the data* student\_first\_name = input("What is the student's first name? ")  
 if not student\_first\_name.isalpha():  
 raise ValueError("The first name should not contain numbers.")  
  
 student\_last\_name = input("What is the student's last name? ")  
 if not student\_last\_name.isalpha():  
 raise ValueError("The last name should not contain numbers.")  
  
 try: *# using a nested try block to capture when an input cannot be changed to a float* student\_gpa = float(input("What is the student's GPA? "))  
 except ValueError:  
 raise ValueError("GPA must be a numeric value.")  
  
 student\_data = {"FirstName": student\_first\_name,  
 "LastName": student\_last\_name,  
 "GPA": float(student\_gpa)}  
 students.append(student\_data)  
except ValueError as e:  
 print(e) *# Prints the custom message* 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')

11. **Run** your script using these input values and **troubleshoot** any errors that occur, until the program runs without errors.

First Name as "Test"

Last Name as "Tester"

GPA as 1.0

12. **Run** your script using this input value and **verify** that the error displays the **"The first name should not contain numbers."** message.

First Name as "Test42"

13. **Run** your script using these input values and **verify** that the error displays the **"The last name should not contain numbers."** message.

First Name as "Test"

Last Name as "Tester42"

14. **Run** your script using these input values and **verify** that the error displays the **"GPA must be a numeric value."** message.

First Name as "Test"

Last Name as "Tester"

GPA as "A"

15. **Locate** the code that writes the table data into a json file.

16. **Add** a **try** statement **before** that code, indent the code, **then add** the following validation and exception handling **after** the code.

try:file = open(FILE\_NAME, "w")  
 json.dump(students, file)  
 file.close()  
 continue  
except TypeError as e:  
 print("Please check that the data is 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 file.closed == False:  
 file.close()

17. **Run** your script and **verify** that the error displays the "Please check that the data is a valid JSON format." message.

**Important:** Now that you know the script runs in PyCharm, it still needs to be tested in OS command/shell. After all, the person using your program should not be expected to install and learn to use PyCharm just to run your program.

18. Open your script file in Windows Explorer or Mac OS Finder.

19. Copy your script file to a folder that will be easy to access, such as in your "**documents/PythonCourse**" folder.

20. **Open** a command shell terminal window.

21. **Navigate** to the location of your script file.

22. **Run** and test the script using the Python interpreter program Python.exe or Python3.exe.

After completing this lab, please watch the following video review:

## Demo/Video: Mod05-Lab03-Review

In this lab, you modified the application to include structured error handling. This is considered an important part of any profession program you create, but as you can see, your code is becoming very complex. When your programs get to this level of complexity, it becomes very important to save your code so that you do not lose all the work you put into it. Let's look at some of the ways professional programmers save and manage their work.

|  |
| --- |
|  |

# Managing Code Files

Saving and sharing code files are essential practices that foster collaboration, code quality, knowledge sharing, and overall efficiency in software development. These practices are critical for both individual developers and teams working on projects of all sizes.

Saving and sharing code files are important practices in software development and collaborative work for several key reasons:

* Collaboration: Many software projects involve multiple developers working together. Saving and sharing code files allows team members to work on different parts of a project simultaneously, increasing productivity and speeding up development.
* Code Reusability: By saving code files, developers can reuse code snippets, functions, or modules in multiple projects. This reduces redundant work and ensures consistency in code across different projects.
* Knowledge Sharing: Sharing code files helps in sharing knowledge and best practices within a development team. Developers can learn from each other's code and improve their skills.
* Error Detection and Debugging: When code is shared, it can be reviewed by peers, which helps in detecting errors, bugs, or potential issues early in the development process. This leads to higher code quality.
* Documentation: Code files often serve as a form of documentation. They provide insights into how a software component or system works, making it easier for developers to understand and maintain the code over time.
* Version Control: Saving and sharing code files are integral to version control systems like Git. These systems track changes to code over time, making it possible to revert to previous versions, collaborate on features, and resolve conflicts.
* Backup and Disaster Recovery: Storing code files in a central location, such as a version control system or network share, acts as a backup. This helps in recovering code in case of hardware failures, data loss, or other disasters.
* Code Reviews: Code sharing facilitates the process of code reviews. Peer reviews provide an opportunity for developers to assess code for quality, security, and adherence to coding standards.
* Remote and Distributed Teams: In today's global work environment, many development teams are distributed geographically or work remotely. Sharing code electronically allows teams to collaborate effectively regardless of their physical locations.
* Open Source Development: In open-source software development, code sharing is fundamental. It enables a community of contributors from around the world to collaborate on projects, leading to innovation and the creation of valuable software.
* Knowledge Transfer: As developers join or leave a project or organization, sharing code files helps transfer knowledge about the project's architecture, design choices, and coding patterns.
* Compliance and Auditing: In certain industries, such as finance and healthcare, there are regulatory requirements for code documentation and auditing. Saving and sharing code files helps organizations meet these compliance requirements.

## Network File Sharing

Developers often work on projects as part of a team. When several developers are working together on a project, it's important to have a central place where they can store and share their code.

A network share is like a shared folder on a computer network. Think of it as a virtual space where multiple people can store and access files. It's a bit like having a shared folder on a cloud drive that only computers within your organization can access. Using network shares predates using a cloud drive and is still in use at many companies.

Using network shares, developers can collaborate more effectively because they can easily access and modify the same set of files. It's a simple way to organize way to work on software projects, especially when working as part of a team.

### Using a Network share

Here is a basic example of how to save a code file (e.g., text.py) to a network share on a Windows computer. Keep in mind that network share paths can vary depending on your network configuration, so you should replace the placeholders with the actual network share path you have access to.

To save a file to a network share using the terminal on both Windows and macOS, you can use the copy (Windows) or cp (macOS) command. You'll need to specify the source file (the file you want to copy) and the destination path (the network share path where you want to save the file). Here are the commands for both operating systems:

Windows (using Command Prompt):

copy "C:\path\to\text.py" "\\server\share\path\text.py"

* Replace "C:\path\to\text.py" with the full path to your local text.py file.
* Replace \\server\share\path\ with the network share path where you want to save the file.
* Make sure to enclose paths with spaces or special characters in double quotes.

macOS (using Terminal):

cp "/path/to/text.py" "/Volumes/share/path/text.py"

* Replace "/path/to/text.py" with the full path to your local text.py file.
* Replace "/Volumes/share/path/" with the network share path where you want to save the file.
* macOS typically mounts network shares under /Volumes/, and then you specify the path within the share.

For both Windows and macOS, you must ensure that you have the necessary permissions to write to the network share, and you may need to authenticate if prompted.

**Note:** This is hard to demonstrate without a complex setup, but we wanted you to know about it since many organizations still use network shares in development. If your organization is one of those; contact your IT department for more information.

## Cloud File Sharing

Cloud file sharing has become the modern what to share files or folders. Like using network shares, it allows multiple users or collaborators to access and collaborate on the same files or documents from different devices and locations. This can be particularly useful for team collaboration, sharing documents with clients, or distributing files publicly.

Cloud storage refers to the practice of storing digital data, files, documents, or any type of information on remote servers hosted by cloud service providers. These servers are typically accessible via the internet. Cloud storage services often offer features such as data redundancy, scalability, and accessibility from various devices and locations. Some popular cloud storage providers include Google Drive, Dropbox, Microsoft OneDrive, and Amazon S3. Users can upload, download, and manage their files stored in the cloud.

**Important:** Keep in mind that when you company stores its files on a cloud service, it is actually using the network shares of another company's computers via the Internet. This means that the hosting company has access to your files. In most cases this is not a problem, since the hosting company can be trusted to respect your privacy, however, this does represent a risk and may even be prohibited by law with some types of files and data.

## GitHub

GitHub is often described as a cloud-based platform for code hosting and collaboration rather than a traditional "cloud file sharing service." However, GitHub's functionality includes elements of cloud-based file sharing within the context of software development. Here's why GitHub can be considered a cloud-based file sharing service:

* File Hosting: While GitHub's primary focus is on code, it also allows the hosting of various file types, including documentation (e.g., Markdown files), images, and binary assets. These files can be shared and accessed via the GitHub platform.
* Code Repository Hosting: GitHub allows developers to host and manage code repositories in the cloud. These repositories can contain code files, configuration files, documentation, and other project-related files. In essence, GitHub serves as a cloud-based storage system for code files.
* Version Control: GitHub is known for its robust version control system, primarily based on Git. Developers can use Git to track changes to their code files over time. This includes creating, editing, and deleting code files. These changes are saved and can be shared with collaborators, making it a form of file versioning and sharing.
* Collaboration Features: GitHub offers collaboration features such as pull requests, code reviews, and issue tracking. These features involve sharing code files, reviewing changes, and discussing them with collaborators. It's similar to collaborating on shared documents in a traditional file sharing system.
* Access Control: GitHub provides fine-grained access control over repositories. Repository owners can specify who has read-only access and who has write access. This access control extends to code files and other project-related assets, allowing users to share and collaborate on files securely.
* Pulling and Pushing Files: Developers can clone repositories to their local machines, edit code files, and then push those changes back to the cloud-based repository. This involves sharing code files between local and cloud-based storage.
* Cloud-Based Access: GitHub provides web-based access to code repositories and files. Developers can access their code and collaborate with others from anywhere with an internet connection, making it a cloud-based file sharing and collaboration platform.

While GitHub primarily focuses on code hosting and version control, it encompasses many features that facilitate the sharing, collaboration, and management of code and related files in a cloud-based environment. This makes it a powerful tool for developers and teams to work on software projects collaboratively.

## Using GitHub's Website

To get started using GitHub, you will need an account. This process is like creating most web software accounts and is tied to an email account.

You must use a real email account, since GitHub will send you a validating email to that account before you are allowed access to the site. This email address can be an organization account, like @uw.edu, or your own personal account. You can even set up multiple user accounts, one per email address.

A screenshot of a computer

Description automatically generated

Once you have an account, you need to create a repository. You can think of a repository as a set of shared folders where your files are stored and managed through GitHub's web server. To create a repository, you need to supply its name and configure it to be either private or public. If the repository is public, it can easily be seen and shared with other people. One account can have many repositories, each one used for a different project of purpose.

To create a new repository, you:

1. Navigate to the upper-right corner of the page where you'll find a drop-down menu.
2. From the drop-down menu, choose the option that says "New repository."
3. You'll be prompted to enter a simple and easy-to-remember name for your repository.
4. Optionally, provide a description for your repository to give others more context about your project.
5. Select the visibility setting that suits your needs for the repository. This determines whether it's public, private, or limited to specific people.
6. Ensure this option is selected if you want to include an initial README file. (If you don't the repository will be empty and will not be created, unless you add some other files.)
7. Finally, click the "Create repository" button to complete the process and create your new repository.

A screenshot of a computer

Description automatically generated

Once you create the repository, you upload files to it. These files can be viewed, downloaded, and modified by anyone with a link, so be careful what you upload!

**Notes:**

* You should include a starting readme file in the repository. If you don't you will not see the repository folders materialize until you start loading files into it via the Git command-line tool.
* In this course, we use the Web UI to make using GitHub more accessible; however, using the Git command tool is the preferred method used in the industry.

## Demo/Video: Demo04 - Using The GitHub Website

**Exercise**: Take a minute or less to write down a one or two sentence answer to the following question as if you were answering a question from a coworker or interviewer: **What is GitHub?**

# Mod05-Lab04: Saving Files to GitHub

|  |
| --- |
|  |

In this lab, you will create a GitHub repository and practice uploading a file from your computer into that repository. You will do so using GitHub's web interface using your Mod05-Lab03 script file.

**Important:** GitHub changes the look and feel of their website often. This means that the instructions here may not exactly match what you see on the website. This will be true of most of the tutorial videos and articles you find on the internet and is just an unfortunate part of being in the IT industry. So, if something looks different, try your best to figure out what the "button" or "menu option" has moved to. Of course, if you still cannot find it use the module05 discussion board for help.

**Step 1: Create a repository**

1. **Open** a web browser and navigate to <https://github.com/>.

A screenshot of a computer

Description automatically generated

2. **Select** the "Sign up" button and **create** a new GitHub account (as needed). You can use an existing account if your want, but we recommend you create a new account specifically for the course.

**Important: GitHub requires a valid email account for your login.** You may **use your UW email account** or a **new made-up email account** with a made-up name **if you are concerned about security**. If you need more safeguards than that, please talk to your instructor.

3. **Navigate** to the upper-right corner of the page and **Choose** the option that says "New repository."

4. **Enter** **"*Python110" with the quarter and year you are taking this course*** as the name for your repository.

5. **Enter** this description for your repository, "This repository stores the files from my introduction to programing with python course."

6. **Select** the **Public** visibility setting so that other people can access your files.

**Note:** This will only apply to this repository. You can have others that are private if you wish, but your instructor, instructional assistant, and other students in this course all need access to this repository until the course ends.

7. **Ensure** that the "Add a README file" is selected.

8. Finally, **click** the "Create repository" button to complete the process and create your new repository.

**Step 2: Adding your code file to the repository**

9. Navigate to the home page on GitHub.com.

A screenshot of a computer

Description automatically generated

10. **Select** your newly created repository from the Top Repository list. (If you have a lot of existing repositories, use the search box.)

A screenshot of a computer

Description automatically generated

11. On the repository page, just above the list of files in your repository, **locate and select** on the "Add file" dropdown menu. From the dropdown options, **select** "Upload files."

A screenshot of a computer

Description automatically generated

12. **Select** the specific files or folders you want to upload, by either dragging and dropping them into the designated area or selecting "choose your files."

13. **Enter** "Added my Mod05-Lab03 code file to repository" in the "Commit changes" message field.

14. **Select** "Commit changes" to upload your file and save you work.

In this lab, you saved your work to a GitHub repository. From there you can share and manage the file using the web interface. Later your will learn how to work with the repository directly from PyCharm, but for now let's complete our study of the module 5 topics before moving on module 6, where you will learn more ways of organizing and managing complex code.

|  |
| --- |
|  |

# Summary

In this module, we looked at .

At this point, you should try and answer as many of the following questions as you can from memory, then review the subjects that you cannot. Imagine being asked these questions by a co-worker or in an interview to connect this learning with aspects of your life.

* What is the difference between a List and a Dictionary?
* What is the between an Index and a Key?
* How do you write data to a file from a Dictionary?
* How do you read data from a file into a Dictionary?
* What is a JavaScript Object Notation (JSON) file?
* What does Python's json module do?
* What is Structured Error Handling?
* Why is error handling using Try-Except recommend?
* What are two common locations for storing and sharing code files?
* What is GitHub, and why is it used?

When you can answer all of these from memory, it is time to complete the module's assignment and move on to the next module.