

Name: Stephen Jenkins

Date: September 06th, 2023

Course: FDN 110 A: Foundations of Programming – Python

Assignment: Asgmt-07

GitHub Link: <https://github.com/swjenk87/IntroToProg-Python-Mod07>

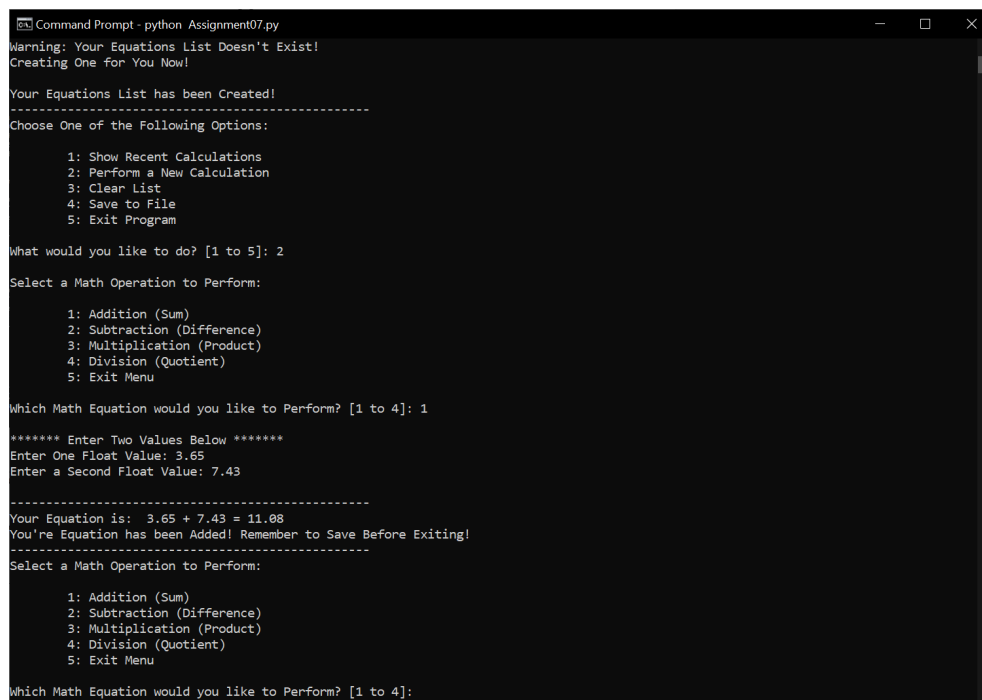
GitHub Website: <https://swjenk87.github.io/IntroToProg-Python-Mod07/>

The Simple Calculator – Error Handling & Pickling

Introduction

Module 07 explored, in more depth, concepts that were touched on briefly in previous lessons including, writing to files, pulling data from files, and handling errors in a more structured, easy to understand way. Though, this week's assignment was more open to interpretation by the students, writing their own functions, and logic, as long as it demonstrated the above-mentioned. The newest of the concepts involved writing data, not in a human-readable, object format, but in binary, using ".dat" files. That, coupled with examples of structured error-handling encompassed the major goals required from the class.

The last major deliverable is the re-formatting of the assignment document into the GitHub markdown language, through the use of a website page. This content will be shared via the discussion board and demonstrates how other programmers and developers store and share information within GitHub.



```
Command Prompt - python Assignment07.py
Warning: Your Equations List Doesn't Exist!
Creating One for You Now!

Your Equations List has been Created!
-----
Choose One of the Following Options:

1: Show Recent Calculations
2: Perform a New Calculation
3: Clear List
4: Save to File
5: Exit Program

What would you like to do? [1 to 5]: 2

Select a Math Operation to Perform:

1: Addition (Sum)
2: Subtraction (Difference)
3: Multiplication (Product)
4: Division (Quotient)
5: Exit Menu

Which Math Equation would you like to Perform? [1 to 4]: 1

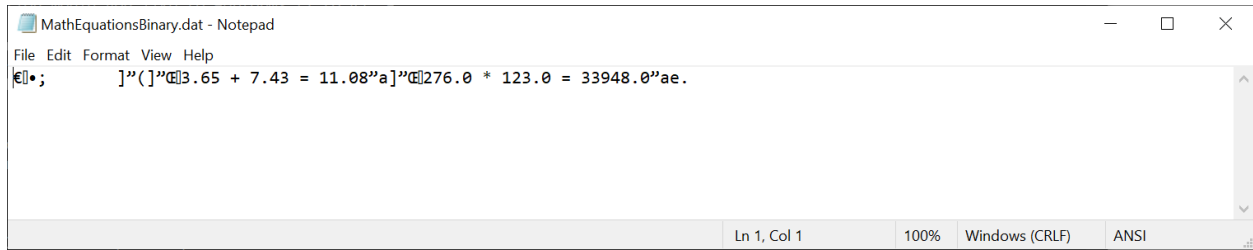
***** Enter Two Values Below *****
Enter One Float Value: 3.65
Enter a Second Float Value: 7.43

-----
Your Equation is: 3.65 + 7.43 = 11.08
You're Equation has been Added! Remember to Save Before Exiting!
-----
Select a Math Operation to Perform:

1: Addition (Sum)
2: Subtraction (Difference)
3: Multiplication (Product)
4: Division (Quotient)
5: Exit Menu

Which Math Equation would you like to Perform? [1 to 4]:
```

Figure 1: The desired result of this assignment



NOTE: Not every combination has been QA'ed. If used in order, as the assignment dictated, the functionality works as intended.

Required Resources and Tools

As has been covered in the previous weeks there were new tools and resources taught that provided additional assistance in creating and executing the weekly assignment. Some of the new concepts are covered in more detail below, with examples from the final, written assignment. Topics covered in the previous weeks will not be included, to focus on the new content from Module 07.

Pickling

Up to this point students have focused on writing out data to text files using human-readable, object-based formats. However, as data becomes larger, and is utilized in more scripts the need will arise for smaller, more compact file and data types that are easier to transport and access.

This is where binary data comes in, using “Pickling” in Python. Pickling is a way of serializing and deserializing data, via the “Pickle Module”, which can be loaded into a Python script. These binary protocols not only make moving data around more efficient but contain everything required to reconstruct the data in other places.

```
14 # The Pickle module implements binary protocols for serializing and de-serializing
15 import pickle
```

Figure 3: Importing the Pickle Module

The “Pickle Module” contains all the necessary functions to pack and unpack these byte streams like “pickle.dump()” which serializes the object structure and “pickle.load()” which deserializes.

```
42 obj_file_data = pickle.load(obj_file)
43 for line in obj_file_data:
44     equation_list_rows.append(line)
```

Figure 4: Example of deserializing a Python object structure

This binary data can be written to a file, similar to how students write to text files in other modules. However, instead of “.txt” the binary streams must be saved to “.dat” files.

With this in mind users must utilize variations of previous writing, reading and appending commands when writing to these “.dat” files. In the past students were trained to use the “open()” function, then

call a file, and then what action should be performed using one single letter: “a” for append, “w” for write, and “r” for read. To ensure the correct file type is used and is being accessed users must now include a “b” to indicated binary.

```
40 equation_list_rows.clear() # clear current data
41 obj_file = open(file_name, "rb")
42 obj_file_data = pickle.load(obj_file)
```

Figure 5: Reading a binary file

Pickling Resources

- **Python Docs – Pickle**
 - <https://docs.python.org/3/library/pickle.html>
 - Definition - The pickle module implements binary protocols for serializing and de-serializing a Python object structure.
 - Module Interface - To serialize an object hierarchy, you simply call the dumps() function. Similarly, to de-serialize a data stream, you call the loads() function. However, if you want more control over serialization and de-serialization, you can create a Pickler or an Unpickler object, respectively.
- **GeeksforGeeks – Understanding Python Pickling with Examples**
 - <https://www.geeksforgeeks.org/understanding-python-pickling-example/>
 - Definition - In Python, we sometimes need to save the object on the disk for later use. This can be done by using Python pickle. In this article, we will learn about pickle in Python along with a few examples.
 - Pickling without a file - In this example, we will serialize the dictionary data and store it in a byte stream. Then this data is deserialized using [pickle.loads\(\)](#) function back into the original Python object.
 - Pickling with a file.

Structured Error Handling

In previous modules students were introduced to the concept of using exceptions to continue execution of code, even when a disruption occurred through “try” and “except”. This week’s assignment dove deeper into raising custom exceptions and utilizing a number of built-in functionalities to ensure that someone who is accessing a program does not encounter an experience-breaking event, like a traceback error, due to an internal script issue or faulty user input that could crash the application. When this occurs in the real world it can lead to a bad user experience and either hurt trust in the program or prevent individuals from utilizing it further.

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    7 / 0
ZeroDivisionError: division by zero
```

Figure 6: Traceback error example

As defined, an exception is an unexpected event that occurs during program execution, such as dividing by zero. Individuals who have used the built-in calculator in Windows have seen that if they attempt to divide a number by zero, a message will be displayed, letting them know that that operation cannot be completed.

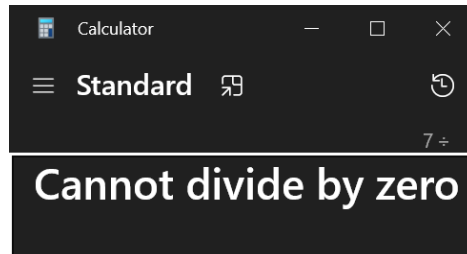


Figure 7: Microsoft Calculator exception message

Errors encountered by users of an application can fall in to two categories, syntax, a misspelling, or misformatted code, and exceptions that are raised when syntax is correct but execution has stopped due to a logistical problem, such as dividing by zero.

To help handle these issues Python has a number of built-in exceptions that can be raised when encountered, giving default readouts, such as the traceback pictured in **Figure 06**. There are also ways of avoiding disruption of the event flow, and instead skipping the execution of certain actions and informing the user of the issue through custom messages. This can be done by different statement types, and through “try” and “except” blocks. The basic concept is similar to if/else statements, where the script attempts to execute one block of script, and if it cannot it will move to the exception, where additional statements can be called.

```
151     try:
152         val_final_div = float(round((val01_div / val02_div), 2)) # Difference rounded to two decimal places
153         print("-" * 50)
154         val_quo_str = str(val01_div) + " / " + str(val02_div) + " = " + str(val_final_div)
155         print("Your Equation is: ", val_quo_str)
156
157         return val_quo_str
158
159     except ZeroDivisionError:
160         print("You Cannot Divide by Zero, Please Enter Different Values!")
161         div_zero = True
```

Figure 8: Try Except Example to mitigate a zero division error

Figure 8 illustrates how a “try” and “except” can be used to help mitigate any potential zero division errors, similar to the calculator example from before. If the user attempts to provide a zero division, they will be alerted they cannot divide by zero and the action will be skipped, asking them for two new values.

So, instead of trying to predict everything a user might provide to the application, the application helps navigate any input that it deems “unacceptable” or “unusable”. This prevents a disruption to the flow of the program and the user experience is maintained.

Structured Error Handling Resources

- **GeeksforGeeks – Python Exception Handling**
 - <https://www.geeksforgeeks.org/python-exception-handling/>
 - Error Definition - Error in Python can be of two types i.e. [Syntax errors and Exceptions](#). Errors are problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which change the normal flow of the program.
 - Difference between Syntax and Exceptions
 - **Syntax Error:** As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.
 - **Exceptions:** Exceptions are raised when the program is syntactically correct, but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.
- **Programiz – Python Exception Handling**
 - <https://www.programiz.com/python-programming/exception-handling>
 - Try Except Block Formatting - For each try block, there can be zero or more except blocks. Multiple except blocks allow us to handle each exception differently.
 - Try, Except, Else - In some situations, we might want to run a certain block of code if the code block inside try runs without any errors. For these cases, you can use the optional else keyword with the try statement

GitHub Pages

Building upon the GitHub repository and page assignments from last week, Assignment 07 requires that the supporting documentation be displayed as a webpage, with a shareable link. This mimics how a lot of industry developers organize and share information regarding their repositories, scripts, projects, etc.

To build these pages GitHub utilizes Jekyll which is a static site generator that uses Markdown and HTML to complete a page layout where information can be stored and displayed back to the user.

Using the tools within GitHub users can select themes, use syntax to format pictures, text, headers and more. There are also a number of plugins and addons that can be used to increase the visual fidelity, interactivity and functionality of the site.

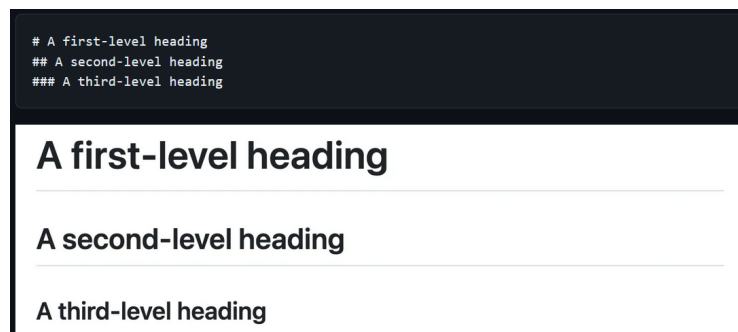


Figure 9: GitHub Page Formatting Example from official documentation

Breaking Down the Application

The new concepts that were utilized in this week's module have been covered with enough detail to get a foundational understanding of how this week's assignment was assembled. The next section will break down the actual code, as it was written, explain the layout, functions that were created and called, and how it was all assembled in the main body of the script.

This week's project is a simple math calculator that allows for addition, subtraction, multiplication and division of two numbers, presented as floats and saved to a list that can be saved to an external file. Due to the complex nature of this script, it would be dozens of pages to cover every line, as was done in previous documents. To that end I have decided to present each section and function, with a description that covers its contribution to the program itself, with an image showing its presentation.

The Header

The header will be added to each week's assignment as it is a best practice in programming to be detail oriented and provide as much information to those accessing the code as possible. This is a collection of fundamental information about the script, its creation and what it does.

- Information included with the Header
 - Title of the Script
 - A brief description of the functionality
 - A change log that includes the creation and any revisions to the script
 - *Note: An additional space was added after the number sign to prevent an underlining format error within PyCharm. Though it was not causing any fatal errors it was improperly setup in previous assignments and has been adjusted from this point on.*

```
1  # ----- #
2  # Title: Assignment 07
3  # Description: Understanding and applying the concepts
4  #               of pickling and error handling within
5  #               Python, through a multi-faceted script
6  #               that read and writes to a file.
7  # ChangeLog (Who,When,What):
8  # SJenkins,2023-08-27,Script framework created for Assignment 07
9  # SJenkins,2023-09-05,Completed coding work to complete Assignment 07
10 # ----- #
```

Figure 10: Header, as shipped with the “What is in Your House” script

Script Segments

There are two main segments to this week's assignment, Functions which are broken up in to multiple classes and the main body which is where functionality and function calls exist, to enable the use of the

program. There are three classes that organize the functions for the “Simple Math Calculator” program. Each one is broken down into its individual parts below.

Functions – Class Processor

Covers the processing of the file data and serialization and deserialization.

Read Data from Binary

This function reads and saves any existing data from a DAT file if it exists, and if it does not exist creates it for the user. There is language built into the function to inform the user of the outcome.

- Two Parameters:
 - file_name
 - equation_list_rows
- Uses a Try/Except block to check for the existence of the data file.
 - Reads from the file if it exists, otherwise writes to create a new file for use.

```
def read_data_from_binary(file_name, equation_list_rows):
    """ Reads data from a file into a list

    :param file_name: (string) with name of file:
    :param equation_list_rows: (list) you want filled with file data:
    :return: (list) list items
    """

    try:
        equation_list_rows.clear() # clear current data
        obj_file = open(file_name, "rb")
        obj_file_data = pickle.load(obj_file)
        for line in obj_file_data:
            equation_list_rows.append(line)
        obj_file.close()
        print("Previous Data has been Loaded!")
        print("-" * 50)

        return equation_list_rows

    except:
        print(str_warning)
        print("Creating One for You Now!\n")

        obj_file = open(file_name, "wb")
        obj_file.close()
        print("Your Equations List has been Created!")
        print("-" * 50)
```

Figure 11: read_data_from_binary Function

Write Data to Binary

Gathers the list that has been generated from the user performing math equations and writes it to a data file, in binary, overriding the previous revisions.

- Two Parameters:
 - file_name
 - equation_list_rows
- Uses the “wb” keyword to write, overriding the binary file with a new version. This was done consciously to demonstrate the different capabilities of the open function.

```
def write_data_to_binary(file_name, equation_list_rows):
    """ List data is saved out to the dat file, overriding previous revisions

    :param file_name: (string) with name of file:
    :param equation_list_rows: (list) you want filled with file data:
    :return: nothing
    """
    obj_file = open(file_name, "wb")
    pickle.dump(equation_list_rows, obj_file)
    obj_file.close()

    return # Return nothing
```

Figure 12: write_data_to_binary Function

Add Equation to List

Gathers the outcome of the math equation action and saves the result to the ever-growing list object. This list is then either trashed, by exiting the program, or written to a file based on user choice.

- Two Parameters:
 - list_of_rows
 - final_equation
- Sets the equation as a list item and utilizes the append functionality to add it to the list table.

```
def add_equation_list(list_of_rows, final_equation):
    """ Add equation to list

    :param list_of_rows: (list) List of equations:
    :param final_equation: (string) Final equation:
    :return: (list) of dictionary rows
    """

    item = [str(final_equation).strip()]
    list_of_rows.append(item)
    print("You're Equation has been Added! Remember to Save Before Exiting!")
    print("-" * 50)

    return list_of_rows
```

Figure 13: add_equation_list Function

Functions – Class MathProcessor

Functions that perform the math operations to generate the user’s list.

Numbers Addition

Performs a sum operation on two values that are provided by the user and outputs a concatenated string showing the calculation and result.

- Two Parameters:
 - val01_sum
 - val02_sum
- Uses the round function to ensure that only up to two decimal places are displayed, keeping the results neater and more organized.

```
def numbers_addition(val01_sum, val02_sum):  
    """ Performs a sum operation on two values  
  
    :param val01_sum: (float) First input float value:  
    :param val02_sum: (float) Second input float value:  
    :return: (string) String operation  
    """  
  
    val_final_sum = float(round((val01_sum + val02_sum), 2)) # Sum rounded to two decimal places  
    print("-" * 50)  
    val_sum_str = str(val01_sum) + " + " + str(val02_sum) + " = " + str(val_final_sum)  
    print("Your Equation is: ", val_sum_str)  
  
    return val_sum_str
```

Figure 14: numbers_addition Function

Numbers Subtraction

Performs a difference operation on two values that are provided by the user and outputs a concatenated string showing the calculation and result.

- Two Parameters:
 - val01_sub
 - val02_sub
- Uses the round function to ensure that only up to two decimal places are displayed, keeping the results neater and more organized.

```
def numbers_subtraction(val01_sub, val02_sub):  
    """ Performs a subtraction operation on two values  
  
    :param val01_sub: (float) First input float value:  
    :param val02_sub: (float) Second input float value:  
    :return: (string) String operation  
    """  
  
    val_final_dif = float(round((val01_sub - val02_sub), 2)) # Difference rounded to two decimal places  
    print("-" * 50)  
    val_dif_str = str(val01_sub) + " - " + str(val02_sub) + " = " + str(val_final_dif)  
    print("Your Equation is: ", val_dif_str)  
  
    return val_dif_str
```

Figure 15: numbers_subtraction Function

Numbers Multiplication

Performs a product operation on two values that are provided by the user and outputs a concatenated string showing the calculation and result.

- Two Parameters:
 - val01_mult
 - val02_mult
- Uses the round function to ensure that only up to two decimal places are displayed, keeping the results neater and more organized.

```
def numbers_multiplication(val01_mult, val02_mult):  
    """ Performs a multiplication operation on two values  
  
    :param val01_mult: (float) First input float value:  
    :param val02_mult: (float) Second input float value:  
    :return: (string) String operation  
    """  
  
    val_final_prod = float(round((val01_mult * val02_mult), 2)) # Difference rounded to two decimal places  
    print("-" * 50)  
    val_prod_str = str(val01_mult) + " * " + str(val02_mult) + " = " + str(val_final_prod)  
    print("Your Equation is: ", val_prod_str)  
  
    return val_prod_str
```

Figure 16: numbers_multiplication Function

Numbers Division

Performs a quotient operation on two values that are provided by the user and outputs a concatenated string showing the calculation and result.

- Two Parameters:
 - val01_div
 - val02_div
- Uses the round function to ensure that only up to two decimal places are displayed, keeping the results neater and more organized.
- Try/Except block to check for division by zero.

```
def numbers_division(val01_div, val02_div):  
    """ Performs a division operation on two values  
  
    :param val01_div: (float) First input float value:  
    :param val02_div: (float) Second input float value:  
    :return: (string) String operation  
    """  
  
    try:  
        val_final_div = float(round((val01_div / val02_div), 2)) # Difference rounded to two decimal places  
        print("-" * 50)  
        val_quo_str = str(val01_div) + " / " + str(val02_div) + " = " + str(val_final_div)  
        print("Your Equation is: ", val_quo_str)  
  
        return val_quo_str  
  
    except ZeroDivisionError:  
        print("You Cannot Divide by Zero, Please Enter Different Values!")  
        div_zero = True  
  
        return div_zero # Back to main with Div Bool
```

Figure 17: numbers_division Function

Functions – Class Input/Output (IO)

The main presentation functions that display menus to the user and except input, to be stored and utilized in the script's functionality.

User Options Menu

Main menu that displays all the functionality options to the user.

- Block print.

```
def user_options_menu():  
    """ Main menu for users to select operations to perform  
  
    :return: nothing  
    """  
  
    print("Choose One of the Following Options: ")  
    print("""  
1: Show Recent Calculations  
2: Perform a New Calculation  
3: Clear List  
4: Save to File  
5: Exit Program  
""")
```

Figure 18: user_options_menu Function

Math Equations Menu

Menu for selection of a math operation to be performed on numbers provided by the user.

- Block print.

```
def math_equation_menu():  
    """ Selection of math operations  
  
    :return: nothing  
    """  
  
    print("Select a Math Operation to Perform: ")  
    print("""  
1: Addition (Sum)  
2: Subtraction (Difference)  
3: Multiplication (Product)  
4: Division (Quotient)  
5: Exit Menu  
""")
```

Figure 19: math_equation_menu Function

Input Menu Choice

Prompts the user to select one of the menu options. This is saved and used in the main body in an if/elif block.

- Returns the user's choice to the main body of the script.

```
def input_menu_choice():
    """ Gets the menu choice from a user

    :return: (string) String Selection:
    """

    menu_user_choice = str(input("What would you like to do? [1 to 5]: ")).strip()
    print() # Formatting

    return menu_user_choice
```

Figure 20: input_menu_choice Function

User Value Input

Prompts the user to provide two float or integer values to be used in the math equations section of the program.

- Uses try/except to ensure that the input provided by the user is a number.

```
def user_value_input():
    """ Gets two float values from the user

    :return: (float) Float value 01, 02:
    """

    while (True):
        try:
            val01 = float(input("Enter One Float Value: ").strip()) # Attempt first float value from user

            break # End loop

        except ValueError:
            print("Not a Number, Try Again")

            continue # Back to the top of the loop

    while (True):
        try:
            val02 = float(input("Enter a Second Float Value: ").strip()) # Attempt second float value from user
            print() # Formatting space

            break # End loop

        except ValueError:
            print("Not a Number, Try Again")

            continue # Back to the top of the loop

    print() # Formatting

    return val01, val02
```

Figure 21: user_value_input Function

Input Math Menu Choice

Prompts the user to select one of the math operation menu options. This is saved and used in the main body in an if/elif block.

- Returns the user's choice to the main body of the script.

```
def input_math_menu_choice():
    """ Gets the math equation menu choice from the user

    :return: (string) String Selection:
    """

    math_menu_user_choice = input("Which Math Equation would you like to Perform? [1 to 4]: ").strip()
    print() # Formatting

    return math_menu_user_choice
```

Figure 22: *input_math_menu_choice* Function

Print Recent Equations

If the main application list is not empty it will print each item back to the user, otherwise it displays a message that it is empty.

- Uses if/else to check for contents in the list.
- Prints the entire length of the list line-by-line.

```
def print_recent_equations(list_of_equations):
    """ Displays

    :param list_of_equations: (list) Each Equation as List items
    :return: nothing
    """

    print("***** Recent Math Equations *****")
    if(len(list_of_equations) == 0):
        print("Your Recent Equations List is Empty, Try Adding Some!")

    else:
        for item in range(len(list_of_equations)):
            print(list_of_equations[item])

    print("*****")
    print() # Add an extra line for looks
```

Figure 23: *print_recent_equations* Function

Main Body

The main body of the script consists of two nested loops and several if/elif blocks that perform a number of different actions based on user input. Each section is broken out with a brief description describing the functionality and what is called.

Load Data

```
# Load any existing data from file
Processor.read_data_from_binary(file_name=binary_file_name_str, equation_list_rows=equation_list)
```

Figure 24: Calls the ability to read from an existing file.

- Checks for an existing file or creates one using the “read_data_from_binary” function.
- Passes two arguments:
 - binary_file_name_str
 - equation_list

Main Menu

```
while(True):
    IO.user_options_menu() # Display the menu options to the user.
    user_choice_str = IO.input_menu_choice() # Return a user's menu choice.
```

Figure 25: Menu and user input calls

- Sets up the main interaction loop, displaying menus and receiving input from the user on which actions to take.

Option 1 – Print Recent History List

```
if user_choice_str.strip() == '1': # Show recent calculations (if they exist)
    IO.print_recent_equations(list_of_equations=equation_list)

    continue # Back to menu
```

Figure 26: Print Equations

- If the user selects option 1 will display any data saved in the global list.

Option 2 – Math Equation Menu

```
elif user_choice_str.strip() == '2': # Perform Math Calculations
    while(True):
        IO.math_equation_menu() # Display the math equations menu
        math_menu_choice = IO.input_math_menu_choice()
```

Figure 27: Math Operations Menu

- If the user selects option 2 sets up an additional while loop allowing the user to perform math equations.

Math Operations

```
if math_menu_choice == '1': # Sum Operation
    print("***** Enter Two Values Below *****")
    val01, val02 = IO.user_value_input() # Receive two values from the user

    sum_final = MathProcessor.numbers_addition(val01_sum=val01, val02_sum=val02) # Additional operation with user input values
    equation_list = Processor.add_equation_list(final_equation=sum_final, list_of_rows=equation_list)

    continue # Back to menu

elif math_menu_choice == '2': # Difference Operation
    print("***** Enter Two Values Below *****")
    val01, val02 = IO.user_value_input() # Receive two values from the user

    sub_final = MathProcessor.numbers_subtraction(val01_sub=val01, val02_sub=val02) # Additional operation with user input values
    equation_list = Processor.add_equation_list(final_equation=sub_final, list_of_rows=equation_list)

    continue # Back to menu

elif math_menu_choice == '3': # Multiplication Operation
    print("***** Enter Two Values Below *****")
    val01, val02 = IO.user_value_input() # Receive two values from the user

    mult_final = MathProcessor.numbers_multiplication(val01_mult=val01, val02_mult=val02) # Additional operation with user input values
    equation_list = Processor.add_equation_list(final_equation=mult_final, list_of_rows=equation_list)

    continue # Back to menu

elif math_menu_choice == '4': # Division Operation
    print("***** Enter Two Values Below *****")
    val01, val02 = IO.user_value_input() # Receive two values from the user
    div_final = MathProcessor.numbers_division(val01_div=val01, val02_div=val02) # Additional operation with user input values

    if(div_final != True):
        equation_list = Processor.add_equation_list(final_equation=div_final, list_of_rows=equation_list)

    else:
        print("-" * 50)

    continue # Back to menu

elif math_menu_choice == '5': # Exit operation menu
    print("Back to Main Menu!")
    print("-" * 50)

    break # by exiting loop
```

Figure 28: Math Operations selection

- Provides multiple options to the user for calculating results for two provided values:
 - Addition
 - Subtraction
 - Multiplication
 - Division
- Provided values are passed to the individual math operation functions where the result is returned as a string and saved to the global list.
- There is an exit criteria that will return the user to the main option menu.

Empty List

```
elif user_choice_str.strip() == '3': # Clear List

    try:
        equation_list.clear() # clear current data

    except:
        print("Your List is Already Empty!")

    continue # Back to menu
```

Figure 29: Clear the global list

- If the user selects option 3 the global list will be cleared, unless it is already empty, which will display a message alerting the user.

Save to File

```
elif user_choice_str.strip() == '4': # Save List to Dat File
    Processor.write_data_to_binary(file_name=binary_file_name_str, equation_list_rows=equation_list)
    print("The Equations Have Been Successfully Saved to the File!")
    print("-" * 50)

    continue # Back to menu
```

Figure 30: Write to file

- Writes the serialized data to an external ".dat" file.

Exit

```
elif user_choice_str.strip() == '5': # Exit Program
    print("Thanks for Using the Calculator!")

    break # by exiting loop
```

Figure 31: Exit Program

- Breaks the main loop, ending the application.

Summary

This module involved two very important concepts, writing to external files and handling errors and exceptions within an application. Attempting to anticipate every move or input a user will make in an application is not realistic. By leaning on built-in and custom exceptions those developing programs can provide avenues for the user experience to be effective and informative. The simple math calculator

program was a way to interact with users, serialize data, write that data to a file and ensure that anything could be entered and handled accordingly with clear, concise messaging.