

Aisha Shafique

05/25/2022

Foundations of Programming: Python

Assignment 07

<https://github.com/Ai-Shaf/IntrotoProg-Python-Mod07>

<https://ai-shaf.github.io/IntrotoProg-Python-Mod07/>

Writing/Reading to File and Exception Handling

Exception Handling

Python will catch syntax errors and often point us toward them even before we execute them. Logic errors we have to sift through ourselves to ensure that our program is doing what we intend it to do. Exceptions are errors that occur when the code is executed, is syntactically correct, but cannot be executed.

Try/Except Blocks

In order to prevent our program from crashing, or, at the very least, being disrupted while running, we can use *try/except* blocks to catch the error and handle it. This can both lead to a smoother flow of the program running, and it allows us the opportunity to provide a more layman-friendly interpretation of the program to the user.

Though there are many websites that can be found to explain how error handling works in python, and I use examples from them in this document where they are clearer, python's documents website (<https://docs.python.org/3/tutorial/errors.html>) provides one of the clearest beginner's explanation of errors and error handling. It begins by showing examples of some common, unhandled errors that emerge with python programming when exceptions occur. This helps a beginner in python start to get acquainted with what these errors are so that they can be expected (with except blocks) and handled for the future.

Each try block can have multiple exception blocks to catch and handle an error that may result from the running of the try block. If the try block results in an error, the program checks to see if the error matches with that in the first except block, if not, it moves on to the next, etc. Once it finds an except block that identifies the error type, it executes that except block and then moves out of the try/except block. In a sense, it is breaking out of the sequence. For this reason, it is always better to keep the error type corresponding with the most general category at the end, so that if the narrowly defined except blocks don't catch the error, the generally defined one can act as a "catch all." However, we always want to try to find narrow definitions of the error to provide more useful information on how to circumvent the error.

See *Figure 1* for an example where the base exception is listed at the bottom as a catch all. Because it is a catch-all, the downside is if we're not vigilant, we might ignore errors that we need to account for elsewhere in our code as well.

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except BaseException as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

Figure 1: Source <https://docs.python.org/3/tutorial/errors.html>

Though so far, this webpage has a useful sequence in starting by explaining common errors and how the try/except block works, it is too brief in its explanation of the try/except/else statement. We know that the else statement follows the except clauses and that it contains code that is executed if the try clause does not raise an exception.

The website https://www.tutorialspoint.com/python/python_exceptions.htm provides a clearer demonstration of how the else block functions. *Figure 2* illustrates how these clauses would look together and the purpose of each. The try block contains the operations we want to conduct, the except clauses try to catch possible exceptions and handle them, and then, if there are no exceptions in the try block, the else statement is executed.

```
try:
    You do your operations here;
    .....
except ExceptionI:
    If there is ExceptionI, then execute this block.
except ExceptionII:
    If there is ExceptionII, then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

Figure 2: Try/Except/Else –Source https://www.tutorialspoint.com/python/python_exceptions.htm

We might question why we don't simply include the else code in the try block? The prior python docs website directly address this: we want to make sure that the exceptions that we are catching are coming from the code that we have designated in the try block. We want to keep code that we haven't planned out exceptions for separate from the code in the try block.

As we saw, the else code only runs if the try code raised no exception. But we may want to have code run regardless of whether the try code ran an exception. This is done by using *finally*. The *geeksforgeeks* website <https://www.geeksforgeeks.org/python-exception-handling/> provides the clearest example of how we use *finally* (see Figure 3 below). If the try block raises an exception, the corresponding except block executes. Regardless of whether the try block raises an exception, the finally block is executed.

Python3

```
# Python program to demonstrate finally
# No exception Exception raised in try block
try:
    k = 5//0 # raises divide by zero exception.
    print(k)

# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

Output:

```
Can't divide by zero
This is always executed
```

Figure 3: Try/Except/Finally—Source: <https://www.geeksforgeeks.org/python-exception-handling/>

An example that ties try/except/else/finally is provided by the Python documents website in Figure 4.

```

>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'

```

Figure 4: try/except/else/finally example –Source <https://docs.python.org/3/tutorial/errors.html>)

Exceptions and Arguments

We can give arguments to error types and save instances of our exceptions and their arguments as variables. *Figure 5* provides a good example, from <https://www.programiz.com/python-programming/exception-handling>, that illustrates both how we can add arguments to exception errors and how we can retrieve both the exception error and argument.

We can raise an error, which forces a specified exception to occur, in the *try* block and after stating it, we can add an argument in parentheses as a string. Then, in our *except* clause, we can state “*except <error name> as <some variable name>*,” followed by a colon. In the statements in that block, we can call on that variable and as shown in the example below, printing that variable will print the argument that we gave the error, rather than the error name itself.

```

>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt

>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument

>>> try:
...     a = int(input("Enter a positive integer: "))
...     if a <= 0:
...         raise ValueError("That is not a positive number!")
... except ValueError as ve:
...     print(ve)
...
Enter a positive integer: -2
That is not a positive number!

```

Figure 5: Exception Arguments and Variables—Source <https://www.programiz.com/python-programming/exception-handling>

Custom Errors

Python also allows user to build subclasses of exceptions that are derived from the built-in python exceptions. Typically, these should come from the Exception class (or subclass).

Once a class is named and defined, then we can use it like any of the prebuilt python error classes. *Figure 6* below illustrates an example of defining a custom error class and executing it.

```

class Networkerror(RuntimeError):
    def __init__(self, arg):
        self.args = arg

```

So once you defined above class, you can raise the exception as follows –

```

try:
    raise Networkerror("Bad hostname")
except Networkerror,e:
    print e.args

```

Figure 6: Custom Error Classes—Source https://www.tutorialspoint.com/python/python_exceptions.htm

Pickling

Understanding pickling through the Python docs webpage (<https://docs.python.org/3/library/pickle.html#:~:text=%E2%80%9CPickling%E2%80%9D%20is%20the%20process%20whereby,back%20into%20an%20object%20hierarchy>) is difficult for a beginner as it jumps right into comparing it with other forms of data conversion such as *marshal* and *json*.

However, the websites <https://www.geeksforgeeks.org/understanding-python-pickling-example/> and <https://www.synopsys.com/blogs/software-security/python-pickling/> provide a clearer description of pickling in python.

Pickling is a way of storing data in a format that makes the data obscure so that it is difficult to understand without reloading it in a certain way. Pickling stores the data in a binary format in a “byte stream”, which is more space friendly and reduces memory usage. It stores the data in a sequence that includes the formatting instructions. Objects also include references to themselves so that they do not have to be repeatedly serialized (<https://www.geeksforgeeks.org/understanding-python-pickling-example/>). This efficiency in space also makes it preferable for transferring over networks. When unpickling, instructions are read from the byte stream to indicate its hierarchy and format and then the data is populated accordingly.

A disadvantage of unpickling is if this is done on a file not obtained from a trusted source, it can unwittingly allow remote access of one’s server to a malicious source. There are ways to account for this—using hmac signatures for instance, which are mentioned in the python docs website as well as this blog—but in general, one should only unpickle from a trusted source (<https://www.synopsys.com/blogs/software-security/python-pickling/>).

Example of Pickling and Structured Error Handling

For this assignment, I write a script that illustrates saving and reading pickled data, as well as using structured error handling in the program.

Writing My Script

My script simply asks the user whether s/he would like to add some data to a list, save data, read it, or exit. This is fairly similar to what we have been doing in assignments thus far.

I use custom functions to separate my data processing from my presentation. In my functions, I have functions for saving and reading data. These not only pickle or unpickle data, respectively, but they also include a try/except block to catch errors. This can happen when no file with such a name exists, as with reading a file, or, more rarely, if there is some obscure problem in the file name or data (essentially the arguments) for saving the data.

```
def read_file(file_name):  
    """  
    Function unpickles and loads data from file_name.  
    :param file_name: name of file to be read  
    :return: file data  
    """  
    try:  
        file = open(file_name, "rb")  
        file_data = pickle.load(file)  
        file.close()  
        print(file_data)  
    except:  
        print("File not found.")  
  
def write_file(file_name, some_list):  
    """  
    Function writes pickled data in some_list to the file file_name.  
    :param file_name: name of file that data is written to  
    :param some_list: list of data that will be written to file  
    :return: nothing is returned  
    """  
    try:  
        file = open(file_name, "wb")  
        pickle.dump(some_list, file)  
        file.close()  
        return 'Success!'  
    except Exception as e:  
        print("Error.")  
        print(e)
```

Figure 7: Pickling and Unpickling with Try/Except blocks

My add data or numbers to list option uses the most functions. It first asks the user to enter two numbers, then asks user which operations s/he wants to conduct on them, and then lastly, it adds the resulting sum, different, product, or quotient to the list of data.

I use try/except blocks to catch if the user does not enter the appropriate integers. *Figure 8* is an example of the more complex of these functions. The function gives the user a menu of math operations, and then runs a loop which while True, “tries” to get the user’s options. Now, if the user chooses an option that cannot be converted into an integer, one of the following exceptions occurs: ValueError or TypeError. If the user does choose an integer but it is not between 1-4, then I raise the Exception with the argument “Choose 1,2,3, or 4.”

```

print(
    """
    1 = Addition
    2 = Subtraction
    3 = Multiplication
    4 = Division
    """
)

while True:
    try:
        choice = int(input("Which operation do you want to do? Choose 1,2,3, or 4:"))
        if choice > 4 or choice < 1:
            raise Exception("Choose only 1, 2, 3, or 4")
        except ValueError as e:
            print("Please choose an integer 1, 2, 3, or 4!")
            print(e)
            continue
        except TypeError as e:
            print("Please choose an integer 1, 2, 3, or 4!")
            print(e)
            continue
        except Exception as e:
            print("There was an error!")
            print(e)
            continue
    else:
        return choice

```

Figure 8: Getting User Input on Mathematical Operations

```

while True: # main script
    print(
        """
        1 = Add numbers to the list
        2 = Save data
        3 = Load data
        4 = Exit
        """
    )
    strChoice = what_to_do()
    if strChoice == "1":
        intA, intB = input_two_int() # capturing the return from the functions into variables
        intOp = math_menu_op()
        intData = math_op(intA, intB, intOp)
        add_to_list(intData, lstInt)
        print(f"You add {intData} to the list {lstInt}")
        continue # return to the beginning of the while loop

    if strChoice == "2":
        write_file(objFile, lstInt)
        print("Data Saved!")
        continue

    if strChoice == "3":
        read_file(objFile)
        continue

    if strChoice == "4":
        strEnd = input("Do you wish to continue? Answer 'y' or 'n'")

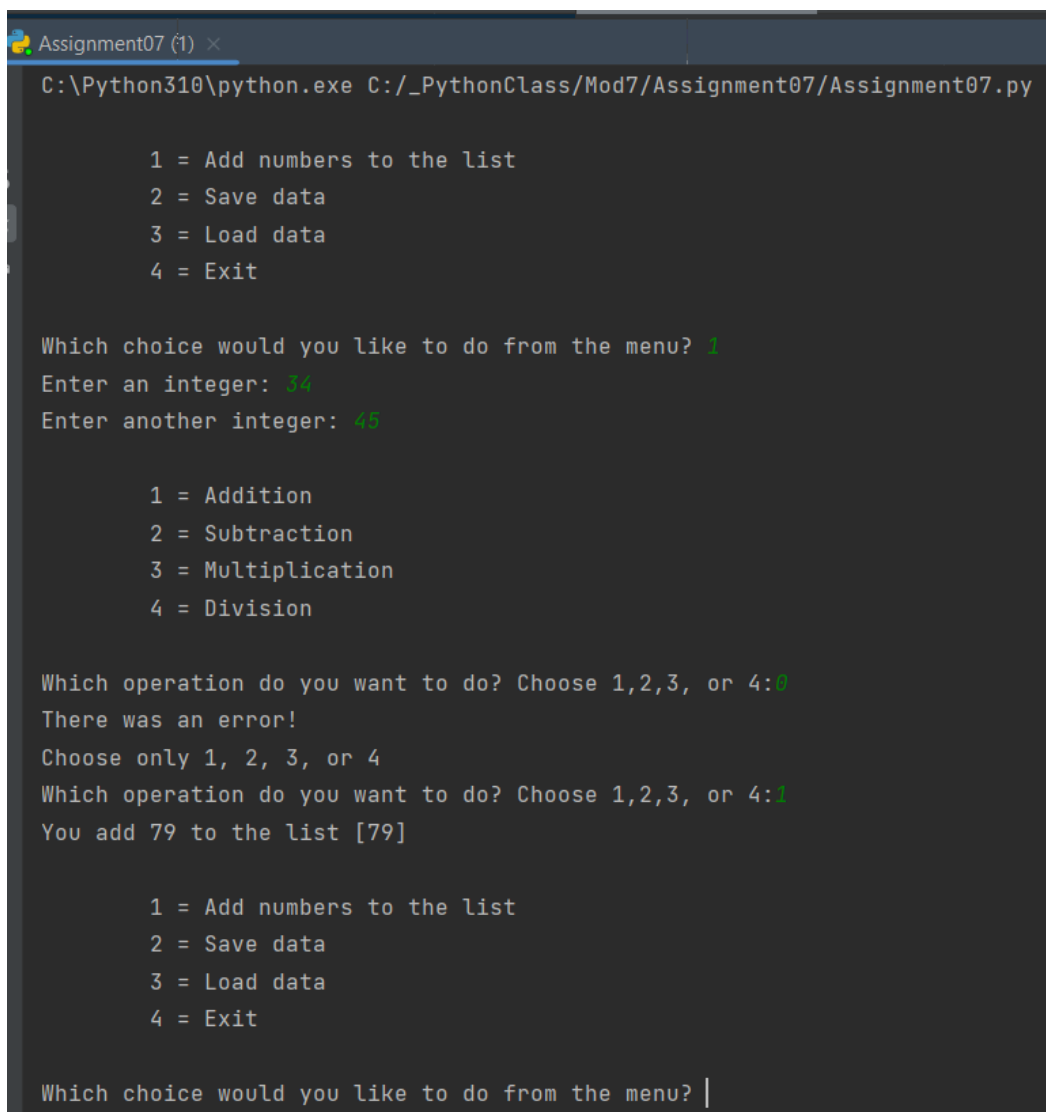
```

Figure 9: Main script of Assignment07.py

For the 4th exit option, the program does what we have been doing thus far: it asks the user if s/he wants to quit, if the user says yes, it breaks out of the main loop, but if the user says no (or anything else), we return to the start of the main while loop.

Output

I ran the script in both Pycharm and Windows OS CMD prompt shell. When running in PyCharm, I realized that my `math_op()` function was not catching the times I entered an integer 1-4, though it caught the errors if I entered something that could not be converted into integers. That was when I modified it by raising a custom error message letting the user know that they need to choose from the given options. Figure 10 below illustrates the program running with that error message showing up.

A screenshot of a PyCharm terminal window titled 'Assignment07 (1)'. The terminal shows the execution of a Python script. The script displays a menu with four options: 1 = Add numbers to the list, 2 = Save data, 3 = Load data, and 4 = Exit. The user is prompted to choose an option. The user enters '1'. Then, the user is prompted to enter an integer, and enters '34'. Then, the user is prompted to enter another integer, and enters '45'. The script then displays a second menu with four options: 1 = Addition, 2 = Subtraction, 3 = Multiplication, and 4 = Division. The user is prompted to choose an operation. The user enters '0'. The script then displays an error message: 'There was an error! Choose only 1, 2, 3, or 4'. The user is prompted to choose an operation again. The user enters '1'. The script then displays the message 'You add 79 to the list [79]'. The script then displays the first menu again. The user is prompted to choose an option. The user enters '1'.

```
Assignment07 (1) x
C:\Python310\python.exe C:/_PythonClass/Mod7/Assignment07/Assignment07.py

    1 = Add numbers to the list
    2 = Save data
    3 = Load data
    4 = Exit

Which choice would you like to do from the menu? 1
Enter an integer: 34
Enter another integer: 45

    1 = Addition
    2 = Subtraction
    3 = Multiplication
    4 = Division

Which operation do you want to do? Choose 1,2,3, or 4:0
There was an error!
Choose only 1, 2, 3, or 4
Which operation do you want to do? Choose 1,2,3, or 4:1
You add 79 to the list [79]

    1 = Add numbers to the list
    2 = Save data
    3 = Load data
    4 = Exit

Which choice would you like to do from the menu? |
```

Figure 10: PyCharm output showing custom error raised

Figure 11 below illustrates the program running in Windows OS console. Here, I have chosen to reload the data. Since I previously added data to the file, it unpickles it and then displays it as part of the `read_file()` function.

```
C:\WINDOWS\system32\CMD.exe - python.exe assignment07.py
Microsoft Windows [Version 10.0.19043.1706]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Herderess>cd C:\_pythonclass\Mod7\Assignment07

C:\_PythonClass\Mod7\Assignment07>python.exe assignment07.py

    1 = Add numbers to the list
    2 = Save data
    3 = Load data
    4 = Exit

Which choice would you like to do from the menu? 3
[-4909, 4.0]

    1 = Add numbers to the list
    2 = Save data
    3 = Load data
    4 = Exit

Which choice would you like to do from the menu? _
```

Figure 11: Assignment07.py running in Windows Console

Summary

In this assignment, we researched on structured error handling and pickling, and then we created our own script that did some of both. Using the basic format of previous assignments, I created a script that gave the user a menu of options and then I defined custom functions that processed the user's choices, whether that be adding information, saving data to file or loading/reading the data to file. Many of those functions used try/except blocks to capture and handle the errors, including the reading and writing functions that unpickled and pickled data from/to file. In one instance, I also used a custom error message as well to ensure the user was selecting from the allowed options. Thus, I was able to encompass pickling and structured error handling into one script.