Justin Calhoun

Professor Summer Rae Elasady

Foundations Of Programming: Python

18 November 2018

Assignment 07 - Exception Handling and Pickling

For this assignment, we were tasked with providing examples of how one might use Exception Handling and Pickling. To do this, I created a small “dummy” program that lets a user experiment with the outcomes of these techniques.

During normal execution, when the Python interpreter runs into some kind of error, it will raise an “exception” and stop processing the script any further. “Exception handling” is the concept of planning for these exceptions and writing your application to behave in a different way when they occur. This is accomplished by placing the block of code you believe may encounter an exception into a try statement, and then using an except to tell the script how to behave when an exception occurs.

In my example, I chose to focus on opening a file, and wrote the following function:



Figure 1: Error handling example function

This function, CharacterCount(), takes a single string as a parameter: a file name. It then “tries” to open the file, and if successful, it will read the contents of the file into a string, and then check the length of that string, or count how many characters are in the file. It returns two items: if successful, the first thing returned is the length as an integer, and the second is the special type None.

However, if it is unsuccessful, it will behave differently. The first except block will execute if there is a FileNotFoundError. This occurs any time the user provides a file name/path that does not exist. Instead of the script ending with the error, though, we “catch” it, and return None for the length, and a custom error string to report back to the user.

The second block checks for a UnicodeDecodeError. This exception is raised when the file isn’t a text file, or if it is a text file, but is not encoded in utf-8. I’m cheating a bit here, because while it’s not exactly correct, my error string asks the user if they’re sure that the file is a text file, implying it’s binary. This is the most likely case for most users that may run this example.

The last except block is a catch-all. It catches any other exception, stores it in the variable e, and then reports it back to the user.

In the main() function, if the user chooses to engage with this example, it will be called by the code in Figure 2.

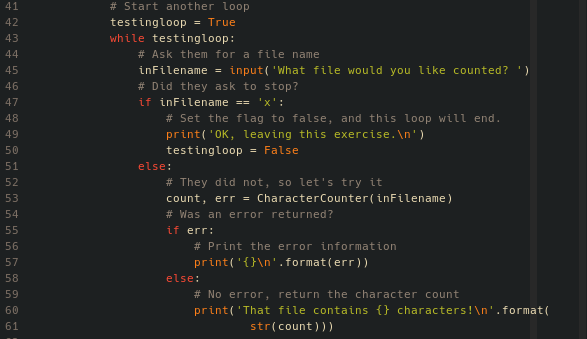


Figure 2: Calling the CharacterCount() function

The script asks the user for a file name. I call the CharacterCount() function with their input string as the argument. If the second variable, err, is returned with anything in it (not None), then I print my custom error and the loop continues. If err was None, we assume that an integer came back, and report the number of characters in the file.

In Figure 3, you can see some of the various responses this function can return, as I test it with a text file, a gzipped tar file, and a file that doesn’t exist.

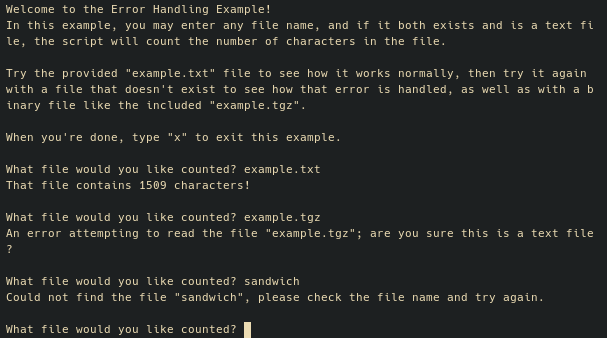


Figure 3: A text file, a binary file, and a file that doesn’t exist

Serialization, or “Pickling”, is the process of taking objects in memory and converting them into a “flattened” binary stream, so they can be stored and retrieved later. This can be handy in some situations. You can pickle objects to prepare them for transmission over a network, or to pass them between running processes in a multi-threaded application. Many highly-interactive applications, like games, use pickling (or similar techniques) to store the entire game state to be later picked up where the user left off. It can also provide very basic obfuscation: instead of storing objects in a human-readable format, if you don’t want your users poking at your stored data in plain text, pickling can “hide” what’s in the file from anyone who might open it and not recognize what it is. It won’t stop a dedicated programmer (or even power user), as they can just unpickle it, but in many cases it may be enough to stop an average user from manually editing application data that might break your program on next load.

The main benefit of pickling, however, is also it’s main drawback. With Python’s pickling, you can store almost any object by serializing it. Importantly, you can pickle functions, classes, or instances of classes. You can also store self-referential objects, such as a list that contains itself as an element. While this is powerful, it’s also quite dangerous: loading pickled objects into memory from a source you do not control or trust (in the IT security sense) can lead to loading and executing malicious code.

The Python documentation calls this out at the top of the page describing the pickle module: “Warning: The pickle module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.” (<https://docs.python.org/3.6/library/pickle.html>). Use pickling with caution, and never unpickle anything that you did not pickle yourself, and thus fully trust the contents.

For my example, I wrote a small function that appends elements to a list:

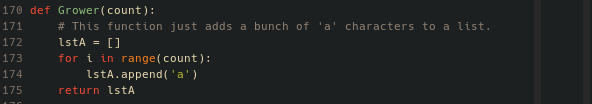


Figure 4: Grower() function

It takes an integer as an argument, and then uses a for loop to append that many items onto a list, which it returns.

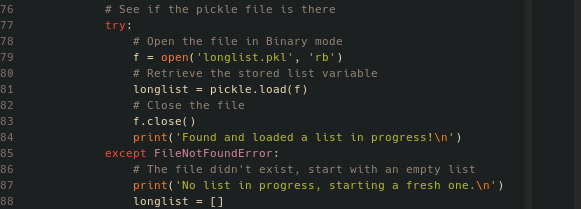


Figure 5: Loading a prior pickle

In the main() function, if the user chooses to interact with the pickling example, I first check to see if I have a pickled file present or not. If so, we unpickle it and load it into memory. If not, we just create a blank list to start growing. Either way, the variable longlist is created.

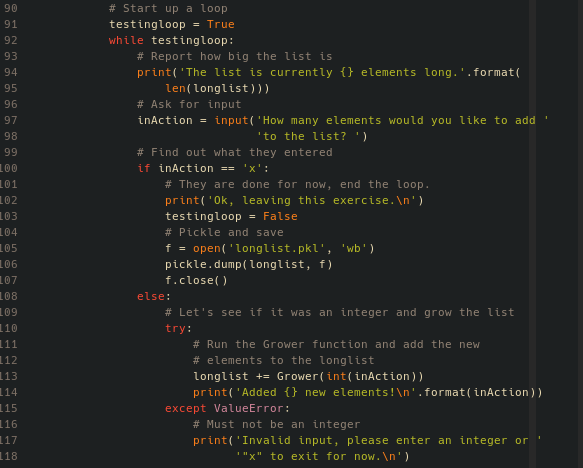


Figure 6: User interaction with the pickle

From here, we start a loop. We ask the user how many elements to add to the list. Grower() is called, and the returned list is added to longlist. When the user tires of adding elements, they can choose to exit. Upon exiting the loop, longlist is pickled and saved to the file longlist.pkl. If they come back later, the pickled list will be opened and loaded back into memory, right where they left off.

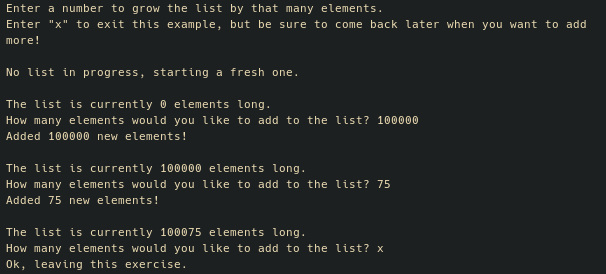


Figure 7: First time loading the pickling example

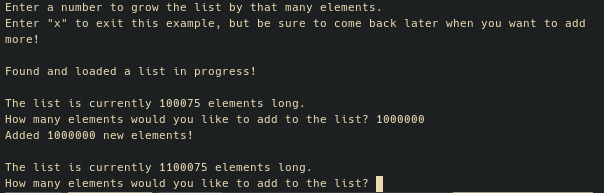


Figure 8: Second time loading the pickling example

While this isn’t a particularly practical example, hopefully it helps demonstrate the power of directly storing and loading objects. Just to see, I timed adding 100,000,000 elements to the list, and it took roughly 16 seconds to create and save the file. I then re-tested, and to load that list from the pickle and re-save it took closer to 6 seconds. While not a very scientific test, there is value to be had in pickling large objects that require heavy computation to create, so that your application can work with that data set immediately upon next execution, instead of having to re-process your source data every time.