11/28/21

Matthew Fuller IT FDN 110 A

https://github.com/mfuller7/IntrotoProg-Python-Mod07/

Assignment 07 Knowledge Document

Introduction

I had never interacted with data in a binary file prior to this lesson. The concept of "obscurity" is interesting and I am curious about the practical application of using this type of file. It was also interesting making the mental shift from working with text file methods to *pickle* methods.

Exception Handling

The first example of Exception Handling uses "Try-Except". Our code is separated into what we would like to happen, the "Try" block, and what we want to happen if an exception occurs, the "Except" block. An example below is used when creating a loop to repeatedly load data from our binary file. The while loop will continually *try* to perform the code in the "Try" block until an exception occurs. At this point the code in the "Except" block is executed. We could even tell the "Except" block to only care about specific exceptions, for example only executing if the error type is the "EOFError".

```
  File "/Users/mfuller/Documents/_PythonClass/Assignment07/Assignment07.
    data.append(pickle.load(file))
EOFError: Ran out of input
```

```
while True:
    try:
        # It will TRY to run the code in the TRY block...
        data.append(pickle.load(file))
        # pickle.load() function loads one line data from file
    except:
        # EXCEPT when it reaches the end of the file, with nothing left to process, where it will
        # raise an "EOFError" exception and perform the code in the EXCEPT block
        break
```

There are also other options to utilize this type of exception handling[1]:

1. **try/except**: catch the error and recover from exceptions hoist by programmers or Python itself.
2. **try/finally**: Whether exception occurs or not, it automatically performs the clean-up action.
3. **assert**: triggers an exception conditionally in the code.
4. **raise**: manually triggers an exception in the code.
5. **with/as**: implement context managers in older versions of Python such as - Python 2.6 & Python 3.0.

Here we are using *if/elif* statements to check if the user input is numeric and avoid errors when performing future math operations. If the input is not numeric we *raise* an exception, with optional message that is displayed to the user.

```
if pickleTaste.isalpha() or pickleTaste.isalpha() or pickleTaste.isalpha():
    # Checking that entered values are numeric, raise exception if not
    raise Exception('ERROR! Please enter numeric pickle rating (1-10). No rating recorded.')
```

Similarly, we can *raise* an exception with a call to a custom error class that we created elsewhere, "PerfectPickleError()". In this example the class takes in an *Exception* as a argument and has one method that returns a string containing an error message we have written that likely makes it more clear to the user why the code did not execute as expected.

```
elif ((int(pickleSmell) + int(pickleColor) + int(pickleTaste)) / 3) != 10:
    # Check to see that user fully appreciates pickles, raises custom exception if not
    raise PerfectPickleError()
```

```
class PerfectPickleError(Exception):
    def __str__(self):
        return 'ERROR! Pickles are perfect. Overall score less than 10/10. Try again...'
```

Pickling

According to the official Python documentation[2],

> Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.

[1] https://www.w3schools.in/python-tutorial/exception-handling/

[2] https://docs.python.org/3/library/pickle.html#data-stream-format

In our case this means that we are using the "Pickle" Python library to move data back and forth from string data (the "Python object hierarchy") to a *.dat* binary file (the "byte stream"). Examples of this are seen in the following code.

```
data.append(pickle.load(file))
pickle.dump(each, file)
```

pickle.load() takes in as an argument an open file object and loads one line of data from it, *pickle.dump()* reverses it and writes data to one line in the file. This is sometimes described as "serializing" and "deserializing".

One advantage of serializing data is to make it more "obscure" as well as more easily transport it across a network. However, one article explains that this also presents a significant draw back in that the "pickled" data cannot be verified before it is "unpickled", potentially being malicious if coming from an unverified source.[3]

Summary

It was good practice to work with binary files and think more about error handling. Expanding one's knowledge of file types and how they can be utilized effectively is likely a handy skill moving forward. Error handling is something I felt I needed more practice with as it seems like an important way to improve application usability and I after this exercise I feel more confident using it in different ways.

Links

https://www.w3schools.in/python-tutorial/exception-handling/

https://www.freecodecamp.org/news/exception-handling-python/

https://stackoverflow.com/questions/41600577/iterating-through-a-list-of-objects-which-is-loaded-by-pickle

https://www.synopsys.com/blogs/software-security/python-pickling/

https://docs.python.org/3/library/pickle.html#data-stream-format

[3] https://www.synopsys.com/blogs/software-security/python-pickling/

Execution Verification