Name: Megan (Megs) Cambra
Date: May 3rd, 2023
Class: IT FDN 110
Github: https://github.com/mcambra56/IntroToProg-Python-Mod07

Assignment 07 – Pickling and Structured Error Handling

Introduction

This simple script demonstrates two new concepts in python: picking and structured error handling. Used to their full potential in more complicated scripts, pickling can help with securing files from prying eyes and decreasing file size, while structured error handling can help find where an error in a code exists and let the user continue to run the rest of the script. These can be very powerful techniques that help turn a hobby coder into a professional.

Pickling

In python, the act of pickling allows an object, such as a list or variable, to be converted into a byte stream to then be stored in a file. Converting to byte stream has several advantages that can especially help when trying to share or store the file: pickling obscures the file's contents so that it cannot be immediately read without unpickling, and pickling may reduce the file's size, especially for larger or complex objects.

While the contents will look obscured to the human eye after pickling, the contents are not encrypted and can be unpickled by anyone who has managed to make it this far in a python class, therefore sensitive content should not be stored this way without further security measures. Additionally, users must be careful when unpickling as they could ne at risk of unpacking a malicious object. Since the user cannot read the file, they will not know what it contains until they unpickle and should only unpickle files from trusted sources.

An example of pickling and unpickling can be found in Figure 1, the first piece of the script. Here, the user will be prompted to input an item and the count of that item, which will be stored and displayed as the 'inventory'. This is then pickled and stored in a file called "InventoryData.dat" and the file is closed. Finally, the file is opened and unpickled before the first row of content is displayed.

```python
# Pickling Demo #
print("1. Let's look at a Pickling Demo")

import pickle  # This imports code from another code file!

# Create an inventory list with item and count from user input
item_name = str(input("Enter an Item: "))
count = int(input("Enter the number owned: "))
inventory = [item_name, count]
print("Here is what you input to the inventory: ", inventory)

# Store the data in binary format with the pickle.dump method
objFile = open("InventoryData.dat", "ab")
pickle.dump(inventory, objFile)
objFile.close()

# Read the data back with the pickle.load or unpickling method
objFile = open("InventoryData.dat", "rb")
objFileData = pickle.load(objFile) # load() only loads one row of data.
objFile.close()

print("Here is the first inventory item and its count from the file: ", objFileData)
print()
```

Figure 1. *Pickling and unpickling script using user input data*

As described above, when running the script, the user will first be prompted to input an item and its count before their input is then displayed back to them. Finally, after the file is pickled and unpickled the user will again see the first row of data in the file. This can be seen in Figure 2 (see Appendix B to see the code run in Windows Command Line). If the file doesn't already exist, python will create the file; however, if it already exists the script will add onto the existing data in the file, in which case the first row of data displayed at the end will not be same row of data the user just entered, as seen in Figure 3. The obscured data file can be seen in Figure 4.

```
C:\_PythonClass\Assignment06\venv\Scripts\python.exe C:\_PythonClass\Assignment07\Assignment07.py
1. Let's look at a Pickling Demo
Enter an Item: Cats
Enter the number owned: 2
Here is what you input to the inventory:  ['Cats', 2]
Here is the first inventory item and its count from the file:  ['Cats', 2]
```

Figure 2. *Running the pickling code with user input*

```
C:\_PythonClass\Assignment06\venv\Scripts\python.exe C:\_PythonClass\Assignment07\Assignment07.py
1. Let's look at a Pickling Demo
Enter an Item: Dog
Enter the number owned: 1
Here is what you input to the inventory:  ['Dog', 1]
Here is the first inventory item and its count from the file:  ['Cats', 2]
```

Figure 3. *Rerunning the pickling code with new user input*

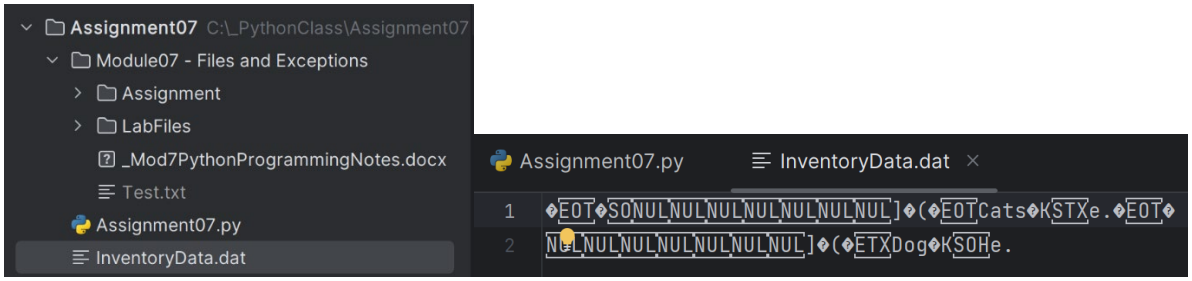

Figure 4a. (left) *Directory showing file created.* ***Figure 4b. (right)*** *File with obscured content after pickling*

Structured Error Handling

When python encounters an error when running code, it returns a complicated error message that is sometimes hard to understand or use to trace down where the error occurred. Python will also end the run as soon as it encounters the error, not allowing the script to finish running through the rest of the code that may be working. Structured error handling allows sections of the code to be run without tripping python errors, and instead allows error handling defined by the coder. This is done in a 'try-except' block, where the code being tested is trapped in the 'try' block, and the 'except' block define how the program should handle errors encountered in the 'try' block. This technique is very helpful in debugging code, especially when new code is introduced.

In the script seen in Figure 5, standard error handling is demonstrated for non-existent file names. This could be the case is the file truly doesn't exist or is not in the same folder as the code and therefore python cannot find it without a path description. This is a common error encountered by python that will usually break the code. However, when inside the 'try' block, the run does not stop and rather turns to the 'except' block to find out how it should be handled. In this case, the user is asked to input a file name which is then read, and the contents are printed. This obviously cannot happen if the file cannot be found. Here, the code in the 'except' block displays an error message explaining to the user that the file cannot be found and then also prints the built-in error message. The standard error handling does not need to be a message and could instead break the code or run a separate piece of code. Many options are available depending on how the coder wants to handle the error encountered.

Figure 6 shows the results of the user inputting a file that does not exist. The user is informed that file cannot be found and then the built-in python error message is displayed. To test the code to make sure it would run properly for an existing file, Figure 7 shows a simple text file that was created containing one row of data. When the code is rerun after the creation of this test file, and that file name is called up, Figure 8 shows that rather than displaying the common error message the contents of the file are displayed.

```python
# Structured Error Handling Demo
print("2. Now let's try structured error handling in Python")

# The piece of code we are trying that will not throw a python error if it fails
# Here we are allowing the user to input a file name
try:
    file_name = input("Input a file name (with extension) that doesn't exist: ")
    f = open(file_name, 'r')
    f_data = f.readlines()
    f.close()
    print(f_data)

# Except statement allows the coder to decide what error to display
except Exception as e:
    print("This file doesn't exist")
    print("Built-In Pythons error info: ")
    print(e)
    print(type(e))
    print(e.__doc__)
    print(e.__str__())
```

Figure 5. *Structured Error Handling script to handle non-existent file with custom message.*

```
2. Now let's try structured error handling in Python
Input a file name (with extension) that doesn't exist: Cats.txt
This file doesn't exist
Built-In Pythons error info:
[Errno 2] No such file or directory: 'Cats.txt'
<class 'FileNotFoundError'>
File not found.
[Errno 2] No such file or directory: 'Cats.txt'

Process finished with exit code 0
```

Figure 6. *Running Structured Error Handling script, where user inputs a non-existent file and a custom error message is displayed.*

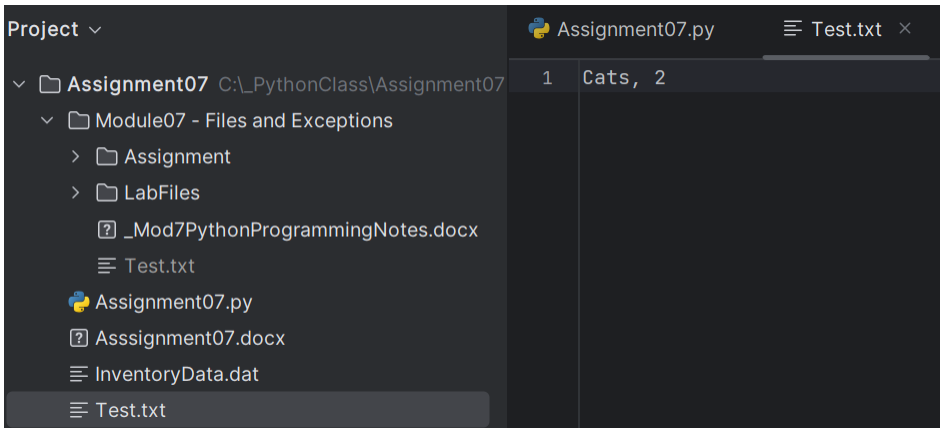

Figure 7. *A text file called "Test.txt" is created in the same directory as the script.*

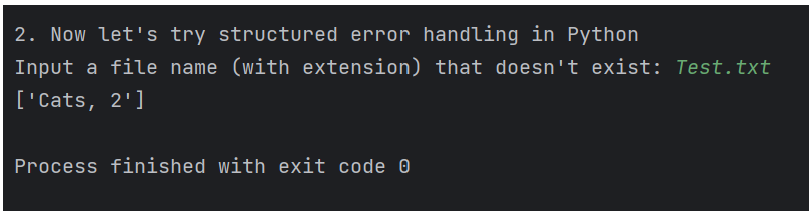

Figure 8. *Running Structured Error Handling script, where an existing file is input and therefore no error message is returned.*

Summary

Pickling and structured error handling are two techniques that can enhance python code and its outputs. Pickling allows a python object to be converted to a byte stream and saved to a file, which obscures the data for human reading and can even help reduce the file size. When unpickling the data users should take caution to only handle files from trusted sources, as malicious objects could be embedded into a pickled file.

Structured error handling allows the code to be run without breaking and triggering the built-in python error messages using ‘try and except’ blocks. A piece of code to be tested is put inside the ‘try’ block, and should it encounter an error python will instead look to the custom error handling inside the ‘except’ block on how to proceed. This could include a custom error message, a break in the code or a completely different path. Structured error handling is very useful when debugging code, especially when new code is introduced by other users.

Appendix A. The entire script

Assignment07.py × InventoryData.dat

```
# ------------------------------------------------------------------------ #
# Title: Assignment 07
# Description: Using pickling to store data in binary format and load it back
#              into the script. Using try-except to demonstrate structured
#              error handling.
# ChangeLog (Who,When,What):
# MCambra,6.3.2023,Created script
# ------------------------------------------------------------------------ #

# Pickling Demo #
print("1. Let's look at a Pickling Demo")

import pickle  # This imports code from another code file!

# Create an inventory list with item and count from user input
item_name = str(input("Enter an Item: "))
count = int(input("Enter the number owned: "))
inventory = [item_name, count]
print("Here is what you input to the inventory: ", inventory)

# Store the data in binary format with the pickle.dump method
objFile = open("InventoryData.dat", "ab")
pickle.dump(inventory, objFile)
objFile.close()

# Read the data back with the pickle.load or unpickling method
objFile = open("InventoryData.dat", "rb")
objFileData = pickle.load(objFile) # load() only loads one row of data.
objFile.close()

print("Here is the first inventory item and its count from the file: ", objFileData)
print()

# Structured Error Handling Demo
print("2. Now let's try structured error handling in Python")

# The piece of code we are trying that will not throw a python error if it fails
# Here we are allowing the user to input a file name
try:
    file_name = input("Input a file name (with extension) that doesn't exist: ")
    f = open(file_name, 'r')
    f_data = f.readlines()
    f.close()
    print(f_data)

# Except statement allows the coder to decide what error to display
except Exception as e:
    print("This file doesn't exist")
    print("Built-In Pythons error info: ")
    print(e)
    print(type(e))
    print(e.__doc__)
    print(e.__str__())
```

Appendix B. Running the code in Windows Command Line

Note that the directory is first changed to the folder where the script lives.

For the pickling demo, since the code was run previously in PyCharm, when the file is unpickled the first row in the file that is now displayed is the one previously saved.