Kyle Odland
June 1, 2020
Foundations of Programming: Python
Assignment 07
https://github.com/oddlandd/IntroToProg-Python-Mod7

Pickling and Error Handling

Introduction

The seventh assignment for Foundations of Programming: Python was to read up on the Python module pickle and on how to perform structured error handling in Python. After doing the research, the assignment was to write a script that would demonstrate how pickling and structured error handling work.

Pickle Research

I began my research into the first of the two topics - pickling - by watching the Module 7 video and reading the course textbook Python Programming for the Absolute Beginner.

The pickling discussion in the course video was relatively short, discussing how binary files can be used instead of text files, and introducing that python has a feature called pickling. The book went through a good example of pickling, explaining that pickling is a good way to store complex data like dictionaries and lists. The example creates three lists, pickles them and dumps them to a binary file, then reads the pickled data back from the file. This shows that the lists themselves are stored to the file, rather than strings as we have been working with previously. The book also discusses shelving as a way to store pickled data. Shelving combines a number of lists together and writes them to a file. It seems to me that we can already combine lists or dictionaries by using them as elements of a list, so I didn’t fully understand the functionality of shelving. The book mentions that using a shelf to simulate a dictionary is more memory efficient than using dictionaries, so perhaps it will be useful for me to use in future programs.

Pickle Coding

My main takeaway from my pickling research, with my current knowledge of python, is that pickling seems most useful to save list or dictionary objects. Pickling them saves the hassle of having to use a for loop to unpack lists and dictionaries to write lines of text to a file, then use a for loop to pack the text back into a list or a dictionary. As an example of pickling, I wrote a script where the user can enter in bills to pay - the bill, the amount, and the date it’s due. These get put in a dictionary with keys “Bill”, “Amount”, and “Due”. Each new input is a new dictionary that is added to the list. The code to accomplish this is very similar to what was done in previous assignments. When the user decides to save the list of bills, the list of dictionary rows is pickled and dumped in a .dat file. Listing 1 shows the function write_to_file(), where the list of bills is pickled and saved in the desired binary file name.

```
def write_to_file(file_name, list_of_bills):
    """ Pickles data and saves it to file

    :param file_name: (string) of dat file
    :param list_of_bills: (list) of dictionary rows
    :return: nothing
    """
    open_file = open(file_name, "wb")
    pickle.dump(list_of_bills, open_file)
    open_file.close()
```

Listing 1: Code to pickle and save the data

To verify that the pickled data is saved as a list object with dictionary rows, I added a function call to the end of the code to read the data from the file, unpickle it and print it. Listing 2 shows the read_from_file() function that accomplishes this.

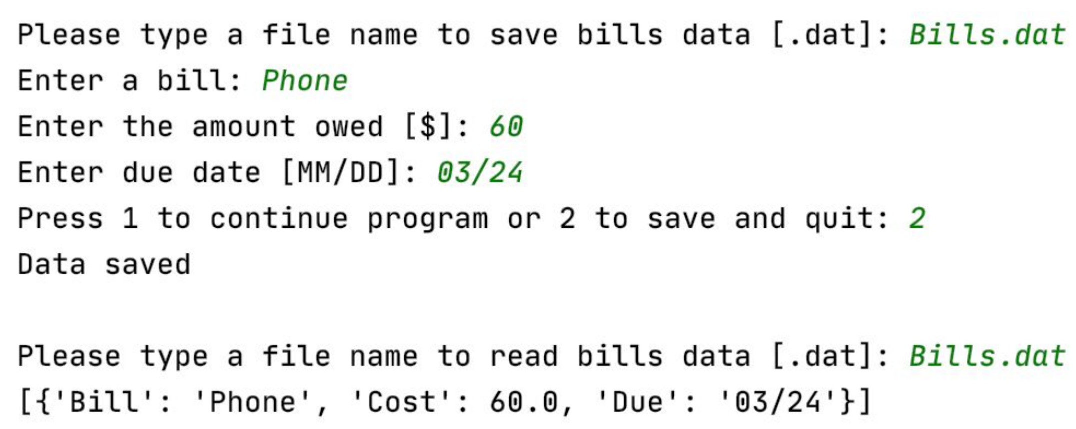
```
def read_from_file(file_name):
    """ Unpickles data from file

    :param file_name: (string) of dat file
    :return: (list) of dictionary rows from file
    """
    open_file = open(file_name, "rb")
    list_of_bills = pickle.load(open_file)
    open_file.close()
    return list_of_bills
```

Listing 2: Code to unpickle the data

Running the Pickle Code

Now that I had functions to pickle and unpickle bills data, I was ready to test out how pickling worked. The main part of my code was just a while loop that allowed the user to keep entering bill information, until they chose to save and quit. Figure 1 shows the code running to test out the pickle module.



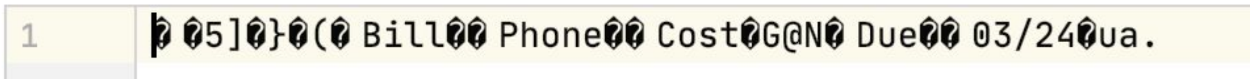
```
Please type a file name to save bills data [.dat]: Bills.dat
Enter a bill: Phone
Enter the amount owed [$]: 60
Enter due date [MM/DD]: 03/24
Press 1 to continue program or 2 to save and quit: 2
Data saved

Please type a file name to read bills data [.dat]: Bills.dat
[{'Bill': 'Phone', 'Cost': 60.0, 'Due': '03/24'}]
```

Figure 1: Testing the bills program to see that pickling worked

The last line just prints the list of bills that was read from the file. The curly braces inside square brackets indicate that the read_from_file function is returning a dictionary inside of a list, showing that pickling allows more complex data to be saved to a file.

Figure 2 shows the data inside the binary file.



```
� �5]�}�(� Bill�� Phone�� Cost�G@N� Due�� 03/24�ua.
```

Figure 2: List of bills inside binary file

Error Handling Research

The second part of the assignment was to research error handling, and to demonstrate error handling in a script. I found the website https://www.datacamp.com/community/tutorials/exception-handling-python (External Link) to be helpful in explaining the use of the try-except block for error handling. Figure 3 shows a good flow chart from the website that illustrates try-except-else-finally.

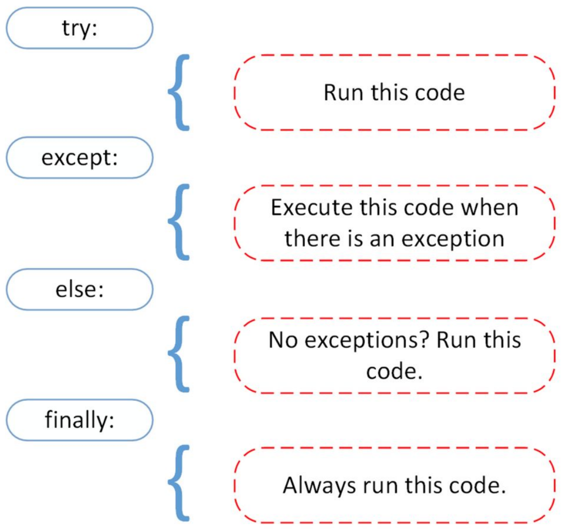

try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

Figure 3: Illustration of try-except block
Source: https://files.realpython.com/media/try_except_else_finally.a7fac6c36c55.png

The website also had good information about specific exceptions that you could call out in the except block, to catch errors that you know have a possibility of arising.

The course Module 7 had good information about raising custom errors and creating custom exception classes. Even if something isn't an error in python, it could lead to an error down the road. By checking with an if statement, you can raise an error if particular code doesn't match your requirements. With custom exception classes, you can add new errors for your code to raise, with custom error messages that you specify.

Error Handling Coding

I decided to put a mixture of the error handling lessons that I had learned into my code. My code asks the user to input the cost of their bill, in dollars. That input statement has a float() callout, so that the cost of the bill is saved as a float, in case I was to use it for something later. In case the user didn't input a number for the cost, I added a try-except statement to my main code that would catch a ValueError. Listing 3 shows the main body of my code for adding bills to the list.

```
while True:
  try:
    lstBills = input_new_bill(lstBills)
  except ValueError:
    print("Please only enter a number for the cost of the bill\n")
    continue
  userChoice = more_bills()
  if userChoice == "1":
    continue
  else:
    write_to_file(strFileName, lstBills)
    print("Data saved\n")
    break
```
Listing 3: Try-Except statement to catch ValueErrors

The other main spot in my code where user input could cause an exception is with the file name. The user has to enter a file name to save the data to, so I made a custom exception class that would cause an error if the file didn't end in .dat. Listing 4 shows my custom exception class FileNotDATError().

```
class FileNotDATError(Exception):
  """ File extension must end with .dat """
  def __str__(self):
    return 'File extension not .dat'
```
Listing 4: FileNotDATError Exception class

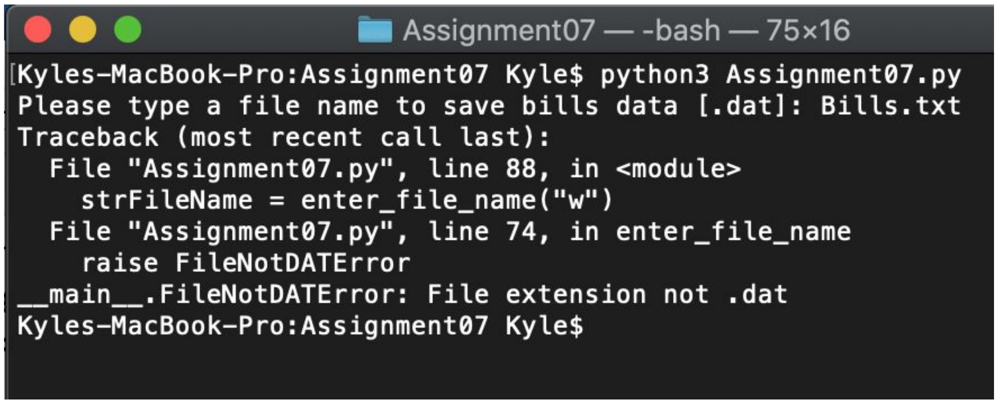
In my function where the user enters the file name, I use an if statement to check if the file_name.endswith() == "dat", and if it doesn't then I raise the FileNotDATError.

My final demonstration of custom error handling is in the reading from file portion of my code. The user enters the name of a .dat file to read from, and I use a try-except-else block to catch the FileNotFoundError if they don't enter an existing file. Listing 5 shows this block of code, to get a file name from the user and read the data from it.

```
# get data from file and print it
while True:
  try:
    strFileName = enter_file_name("r")  # get file name to read from
    print(read_from_file(strFileName))  # print contents of file
  except FileNotFoundError:
    print("File does not exist\n")
  else:
    break  # if no error, break out of while loop
```
Listing 5: Try-except block to catch FileNotFoundError

Running the Error Handling Code

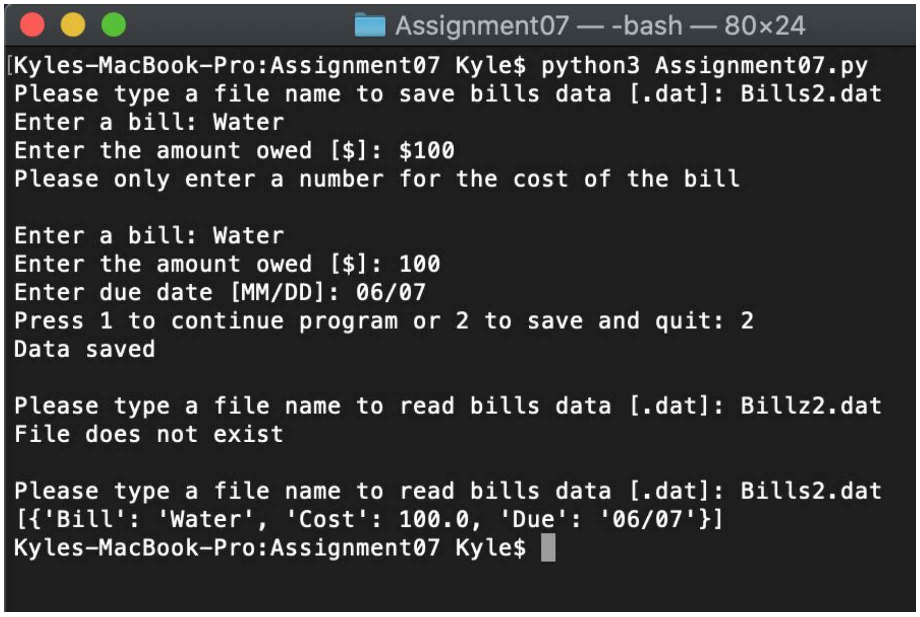
To test out the structured error handling, I ran the code as if the user was making all the worst decisions on what they input. Figure 4 shows the error that comes up if the user enters a file that doesn't end in .dat.



```
Kyles-MacBook-Pro:Assignment07 Kyle$ python3 Assignment07.py
Please type a file name to save bills data [.dat]: Bills.txt
Traceback (most recent call last):
  File "Assignment07.py", line 88, in <module>
    strFileName = enter_file_name("w")
  File "Assignment07.py", line 74, in enter_file_name
    raise FileNotDATError
__main__.FileNotDATError: File extension not .dat
Kyles-MacBook-Pro:Assignment07 Kyle$
```

Figure 4: FileNotDATError shown in macOSX Terminal

I ran it again in the terminal to show other errors that might pop up. Instead of raising the exceptions, I included them in try-except statements, and gave the user a chance to correct their input. Figure 5 shows the code catching a ValueError and a FileNotFoundError, and just asking for new input.



```
Kyles-MacBook-Pro:Assignment07 Kyle$ python3 Assignment07.py
Please type a file name to save bills data [.dat]: Bills2.dat
Enter a bill: Water
Enter the amount owed [$]: $100
Please only enter a number for the cost of the bill

Enter a bill: Water
Enter the amount owed [$]: 100
Enter due date [MM/DD]: 06/07
Press 1 to continue program or 2 to save and quit: 2
Data saved

Please type a file name to read bills data [.dat]: Billz2.dat
File does not exist

Please type a file name to read bills data [.dat]: Bills2.dat
[{'Bill': 'Water', 'Cost': 100.0, 'Due': '06/07'}]
Kyles-MacBook-Pro:Assignment07 Kyle$
```

Figure 5: Structured error handling in List of Bills program

Summary

This module was a lot more open ended than previous modules, and asked us to go and research two different python functionalities. The first was pickling, which is used to save complex data to a file for storage, and seems useful for saving lists or dictionary data into a file. The second was structured error handling, with try-except statements that can give custom error messages so that the user is aware of exactly the issue with their inputs.