Eric Forster

Nov. 29th, 2020

IT FDN 110 A Au 20

Assignment 07

github.com/e4ster/IntroToProg-Python-Mod07

**Pickling and Error Handling**

**Introduction**

This week’s assignment included an open-ended exercise where I created a script that demonstrated how to pickle data, and how to perform error handling. In order to do this, I created a program that asks the user for an ID number, and a first name, stored the data to a binary file, and then printed all of the data in the binary file to the user. Understanding these skills was important because binary file sizes are much smaller and can speed up enterprise applications substantially.

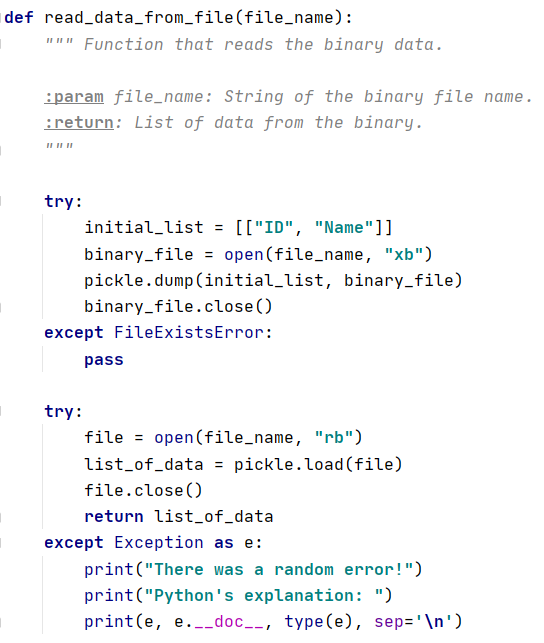
**Step 1: Read Data, Store in List**

The main body of this program was only four lines, however it called functions that were previously defined in the script. This first executable line is below:



***Figure 1.1: Assigning file data to a list.***

In this line, I called a function to read data from the binary file, using our file name as an argument, and then assigning the list it returned to ‘lstCustomer’. Let’s take a look at the ‘read\_data\_from\_file’ function.

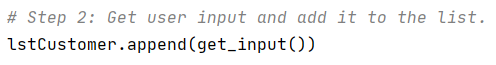


***Figure 1.2: Function to read file data.***

In the ‘read\_data\_from\_file’ function, I started with a try/except block. This tried to create a binary file, and then write an initial list to it with ‘ID’ and ‘Name’. In the except section, I made an exception for ‘FileExistsError’ which would normally pop up if the file already existed. If the file did indeed exist, the function moved to the next try/except block where I opened the file in read mode, loaded the file contents to a list variable, closed the file, and then used that list in my return statement. As an exception for this section, I called a general exception to illustrate a catch-all, but also to illustrate how to print the python information available for the random error. By using the variable ‘e’, I printed the exception, the python documentation, as well as the exception type. This can be useful information for developers to review. Since the function returned a list with at least one element in it from the binary file, this data was assigned to the ‘lstCustomer’ variable.

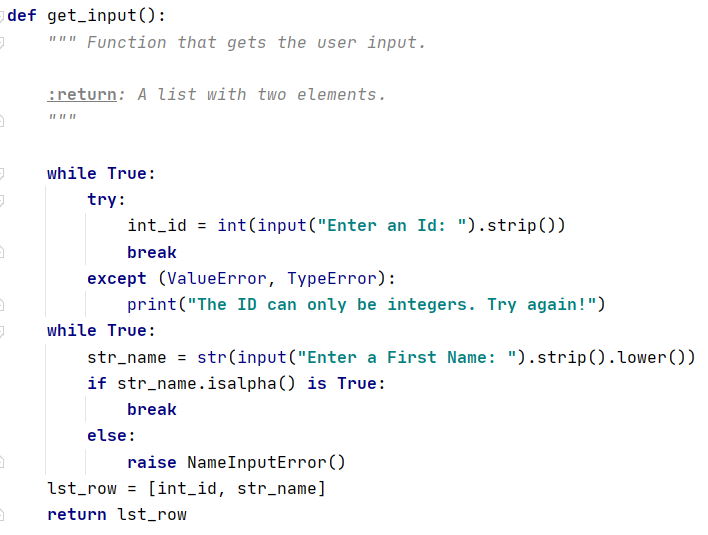
**Step 2: Add User Input to List**

Next in my program, I needed to get information from the user including an ID number, as well as a first name. This information was gathered and put into a list with two elements, and then that short list was added as an element to our ‘lstCustomer’. Here was the executable line:



***Figure 2.1: Get user input and add to list.***

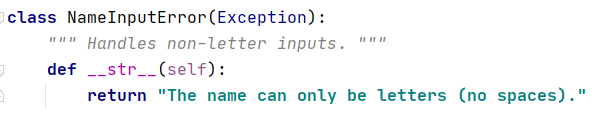
In this line of code, I used the ‘.append’ method from the list class, and for the argument I called a ‘get\_input’ function that I created. Let’s take a look at the ‘get\_input’ function.



***Figure 2.2: Getting the input.***

In the ‘get\_input’ function, I used a couple different examples of error handling. To start, I made a ‘while True’ block, with a ‘try/except’ block nested inside. By asking the user for an ID as an integer data type, it was very possible to run into an error. If there was no error, the script broke out of the ‘while True’ loop. If there was an error, I called two exceptions at once for the interpreter to look through. Once it sees that either ‘ValueError’ or ‘TypeError’ was the issue, it printed the statement provided, and moves back to the top of the loop.

In the next ‘while True’ loop, I asked for user input again, but this time assigned it as string type data. For this block, I demonstrated how a customer exception can be used. I wanted to make sure the string provided only included letters. Using the ‘isalpha’ method, it broke out of the loop if the string only contained letters. If it contained something other than letters, I manually raised a ‘NameInputError’ that I made. See below:



***Figure 2.3: Custom exception.***

This was a custom exception I wrote that inherits from the Exception class. This stops the program and tells the user at what line the issue was brought up, and displays the text I wrote in the return statement. While this wasn’t very elegant in this scenario, it displays the ability to create a custom exception, and raise that exception manually in certain situations. Once the user entered valid inputs, those inputs were stored as a small list, which was then returned.

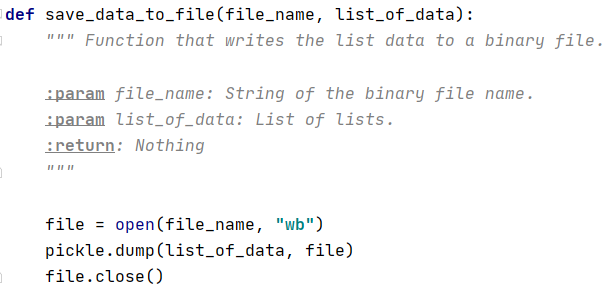
**Step 3: Save the List to Binary File**

Once the user had input new data and it had been appended to our list, it was time to save the updated list to the binary file. See the executable line below:



***Figure 3.1: Calling function to save data.***

I called the function ‘save\_data\_to\_file’ using two arguments. These were my binary file name, and the list of all my data which was stored in memory. Let’s look at the ‘save\_data\_to\_file’ function.



***Figure 3.2: Function that saves to file.***

In the function above, I open the file in the ‘write binary’ mode which would overwrite anything that is currently stored in the file. This ensured I wouldn’t have duplicate items stored in my file. Then, using the ‘dump’ method from pickle, I wrote the list of data to the binary file. Lastly, I closed the file. It is worth noting the use of the open() function within this function I created. By using Python’s built-in function nested inside my own, I was able to open/close the file, and manipulate the data, all while calling one function.

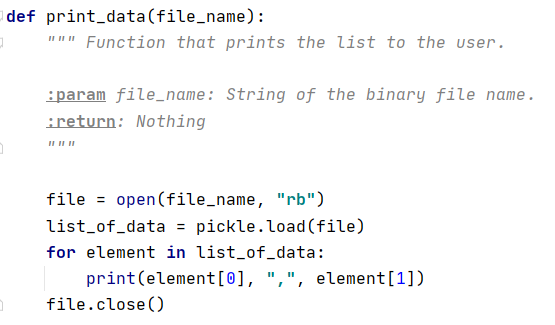
**Step 4: Print Data to User**

For this step, I wanted to gather the most recent data from the binary file, and then print that out to the user in a readable format. Here was the executable line used:



***Figure 4.1: Call to a print function.***

The line above called a custom print function I made, and used the file name as one argument. Let’s look at the ‘print\_data’ function:

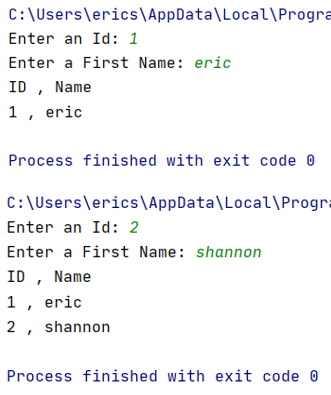


***Figure 4.2: Custom print function.***

This was another case where I used built-in python functions within my own custom function. This allowed me to format that data how I preferred, and then print it out to the user. I opened the file in read mode, then loaded the contents to a list variable. Next, I created a ‘for’ loop and walked through each element in the list. When I went through each high-level element, I printed the two low-level elements inside. Lastly, I made sure to close the file. This concluded the program and it ended.

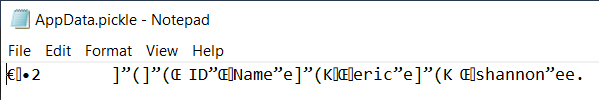
**Step 5: Running the Script**

I will first display the script running twice in PyCharm:



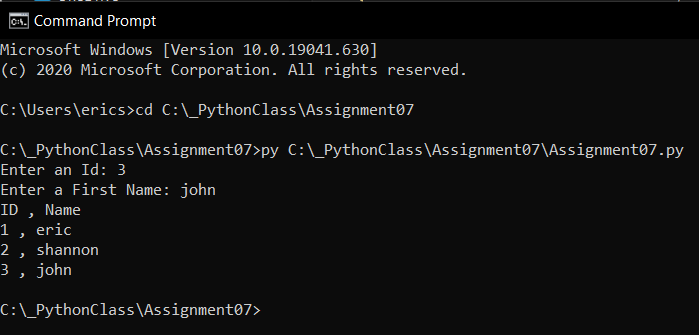
***Figure 5.1: Script running twice in PyCharm.***

Below you can see that the ‘AppData.pickle’ file was created, and data was added to it.



***Figure 5.2: Data stored in the binary file.***

Next you will see the program running from the command prompt.



***Figure 5.3: Script ran from the command prompt.***

As you can see, when I entered the third data set from the command prompt, the data was saved to the file, and then read from the file and printed to the user.

**Summary**

Pickling data is a great way to store data as smaller files. As programs and databases get very large, this might even be required in some cases. I’ve learned the basics, but there is no doubt I need to continue researching everything that the ‘pickle’ module has to offer. In addition, the use of error handling is a necessary part of well performing scripts. Whenever a script deals with input/output, it runs the risk of the user creating an error. Exceptions are a concept that I see myself working on for a long time because there seems to be an aspect of creativity involved. Looking forward to building on these skills!