Eric Forster

Nov. 22nd, 2020

IT FDN 110 A Au 20

Assignment 06

github.com/e4ster/IntroToProg-Python-Mod06

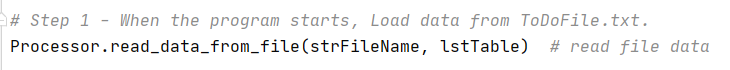
**ToDoFile with Functions**

**Introduction**

For this week’s assignment, I filled in the gaps in a starter script in order to create a new version of the interactive to do list. The bulk of this work was finishing the function definitions, and then calling all of those functions correctly in the main body. The script had a definite structure that correlated with the ‘separation of concerns’ methodology. This made it easy to read and begin my work. This assignment was also a great introduction to classes, and how they differ from functions at a very high level. Since this script is longer than the previous assignments, I’m going to forego showing the header, variables, and main structure, but jump straight into how the functions worked together to make this script run. Like most lines of code, I need to read the script backwards.

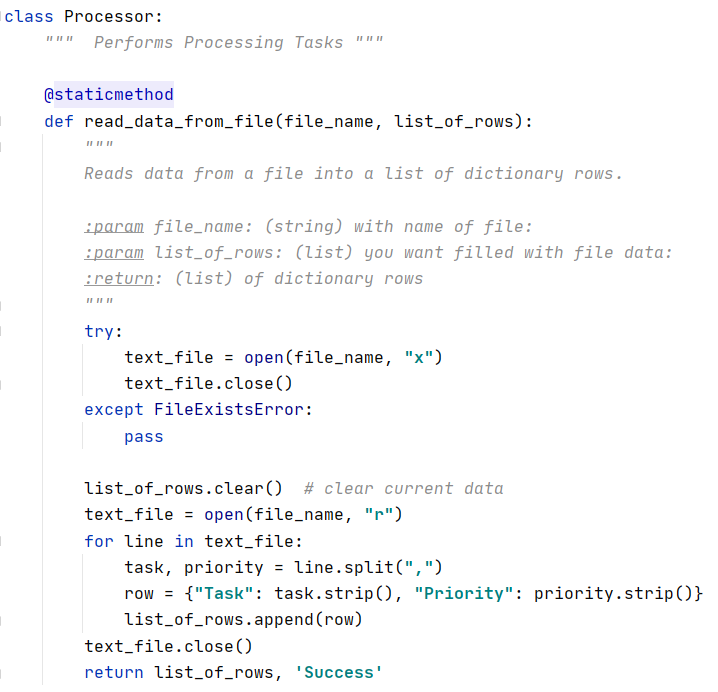
**Step 1: Load Data from Text File**

When this program started running it immediately grabs the current data from the text file and stored it into memory as a list of dictionaries. I wrote a block that first tried to create a file if it doesn’t already exist, and then the next block which grabbed data runs smoothly. Here is the code that called our function:



***Figure 1.1: Calling functions to load initial data.***

From the ‘Processor’ class, I called a method called ‘read\_data\_from\_file’. This method had two parameters, so I used two arguments when I made the call, ‘strFileName’ and ‘lstTable’. These were the name of my text file, and our list of dictionaries. In this first example I will show the class, function definition, and docstring to explain what is going on, but will refrain and just show the meat and potatoes moving forward.

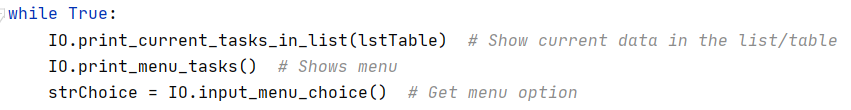


***Figure 1.2: Read initial data from file function.***

First, I created a class called ‘Processor’. The triple quote below the class name is a docstring which describes what the class does. Then I created a static method. From my understanding, this is a function that can be used and called like a normal function, without first creating an object of this class type. Next, I created the function definition with two parameters ‘file\_name’ and ‘list\_of\_rows’. These parameters are used as variables within the function code. In the try/except block, I tried to create the file, and if the file already existed, it passed over the error. This way, the next chunk where I read the file, didn’t have an error the first time the code was ran. For each line in the text file, this code created two variables ‘task’ and ‘priority’ and saved them as dictionary values in a sequence of dictionaries. Since we passed ‘lstTable’ as the second argument in the function call, and the method returned and updated a list for the second parameter, the ‘lstTable’ was updated and ready to manipulate.

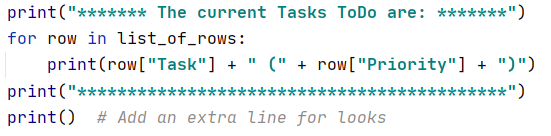
**Step 2: Start Loop, Display Data, Get Menu Choice**

Next, I started a ‘while True’ loop, and then performed all of these actions with just three lines of code. This was made possible by defining these functions earlier in the code.



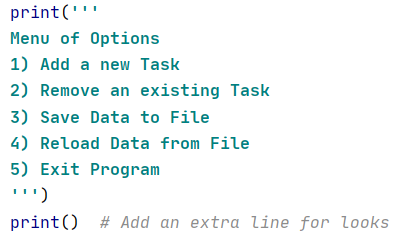
***Figure 2.1: Start of the loop.***

From the ‘IO’ class, I called the method ‘print\_current\_tasks\_in\_list’, and used the argument ‘lstTable’.



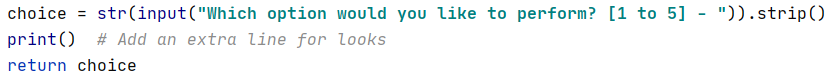
***Figure 2.2: Print current tasks method.***

This method worked through each element in the list of dictionaries, and printed the values stored in each dictionary for the user to see. After this was performed, I called the method ‘print\_menu\_tasks’ from the class ‘IO’.



***Figure 2.3: Print menu method.***

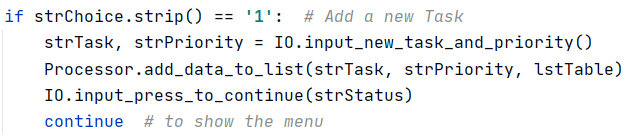
This simply printed the user’s options similar to previous assignments. Once the user knew what the options were, the program called the method ‘input\_menu\_choice’ from the class ‘IO’. Then it stored the returned value from the method as the variable ‘strChoice’.



***Figure 2.4: Input choice method.***

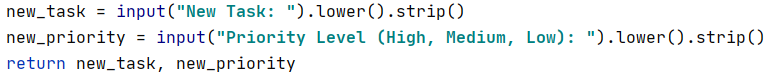
**Step 3: Add a New Task**

When the user chose this from the menu, this is the code that was ran:



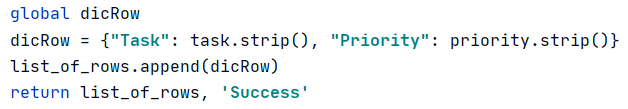
***Figure 3.1: Add a new task function calls.***

First, I called the method ‘input\_new\_task\_and\_priority’ from the class ‘IO’. This method returns two strings, which I assigned to the variables ‘strTask’ and ‘strPriority’ all in one line of code. This method used the input() function to get user input.



***Figure 3.2: Get task and priority method.***

Next from class ‘Processor’ I called the method ‘add\_data\_to\_list’ and used three arguments. These arguments included variables that we assigned values to from the previous line of code.

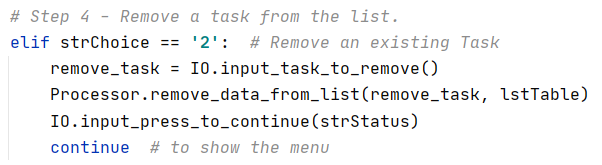


***Figure 3.3: Add data to list method.***

For this method I wanted to use the same variable ‘dicRow’ that was available to the rest of the program. To do this, I declared it as a global variable so that the method recognized it, and could use it. Similar to previous assignments, I created a dictionary of the new task and priority and then added it to our list in memory. The next line created a pause in the program, and waited for the user to hit enter. It was added at the end of each menu choice block, and I won’t cover it moving forward.

**Step 4: Remove a Task**

When the user wanted to remove a task from our list in memory, this is the code that ran:



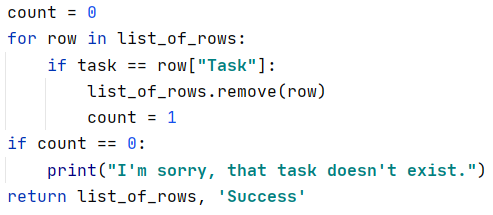
***Figure 4.1: Remove data from list.***

To start, I called the method ‘input\_task\_to\_remove’ from class ‘IO’. This method returned one string and I assigned it to a new variable.



***Figure 4.2: Task to remove method.***

Now that I had user input saying what task was to be removed, I called the method ‘remove\_data\_from\_list’ and used two arguments. One of which was the variable I just created in the previous line.

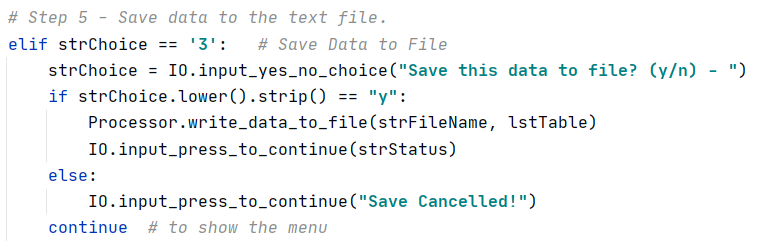


***Figure 4.3: Remove data method.***

This method worked through each element in the list of dictionaries, and checked to see if the user input matched the value of one of the dictionary keys. If it found a match, it removed the element from the list. It then returned the updated list for our ‘lstTable’.

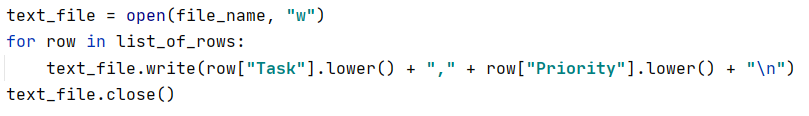
**Step 5: Save Data to File**

When the user chose this from the menu, here is the code that ran:



***Figure 5.1: Save data to file.***

For this block I called the method ‘input\_yes\_no\_choice’ and then assigned the returned value to the variable ‘strChoice’. This method simply used an input function and used the argument as the message the user saw. If the user typed ‘y’ then the next part executed. If not, then the operation was aborted. In order to actually write the data to the file, I called the method ‘write\_data\_to\_file’.

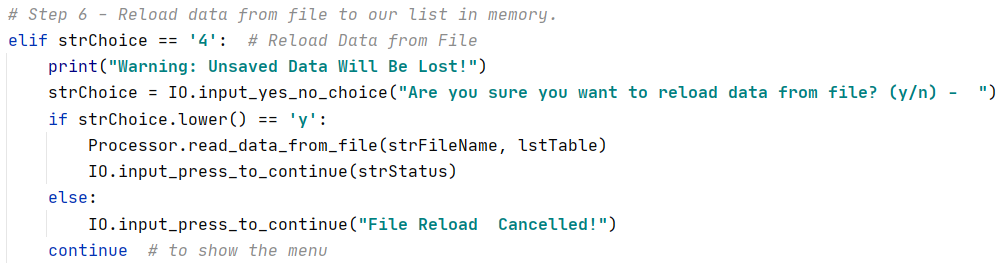


***Figure 5.2: Write data method.***

In the method above, I opened the text file with the ‘write’ argument, which meant it wrote over any data that was previously in the file. Then for each element in our list of dictionaries, the code wrote the value for the key ‘Task’, and the value for the key ‘Priority’, and then did a carriage return. After this was done, I closed the file.

**Step 6: Reload Data from File to List**

If the user wanted to start fresh with the data that was already saved in the text file, they chose this option. In order to do this, the following code was performed:

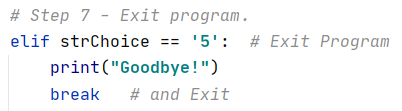


***Figure 6.1: Reload original list from file.***

Notice in this block of code, I used the same ‘input\_yes\_no\_choice’ method, and ‘read\_data\_from\_file’ method, as before. In this process the code cleared the current contents of the list in memory, and then read through the text file appending values to the list. For this option, a warning message was added to make sure this was the option the user intended.

**Step 7: Exit the Program**

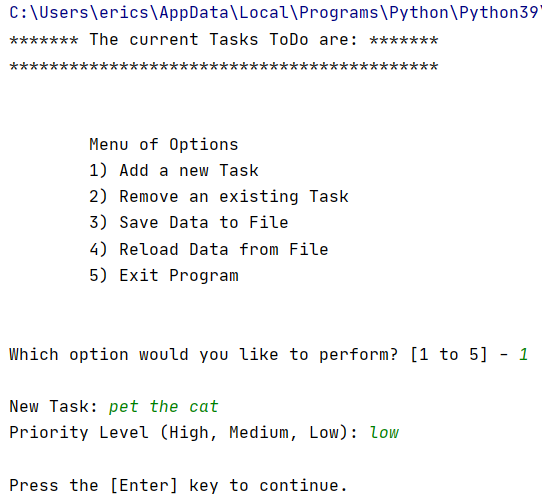
The last section of this program allows the user to exit the program. It does this by running the ‘break’ statement which breaks out of the ‘while True’ loop we started at the beginning.



***Figure 7.1: Exit the program.***

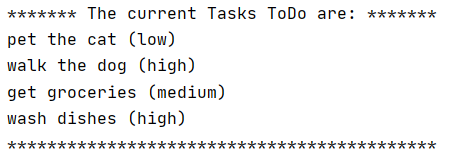
**Step 8: Run the Script in PyCharm / Command Prompt**

In PyCharm, I started by adding four items to my list using option 1:



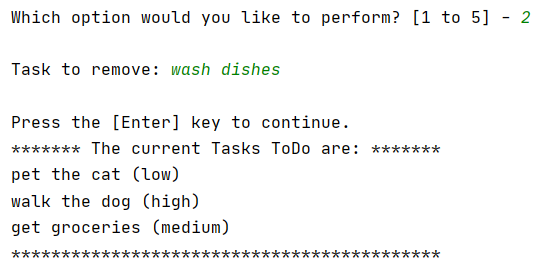
***Figure 8.1: Running option 1.***

Eventually the list in memory go to this point:



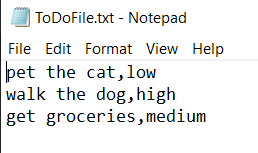
***Figure 8.2: Four tasks printed to screen.***

Here was removing an item from the list. There are only three items:



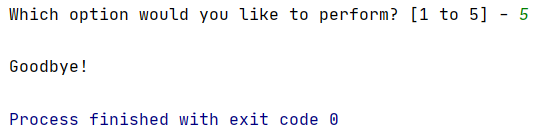
***Figure 8.3: Item removed from list.***

When the user chose the number three, the program saved this data to the text file it created:



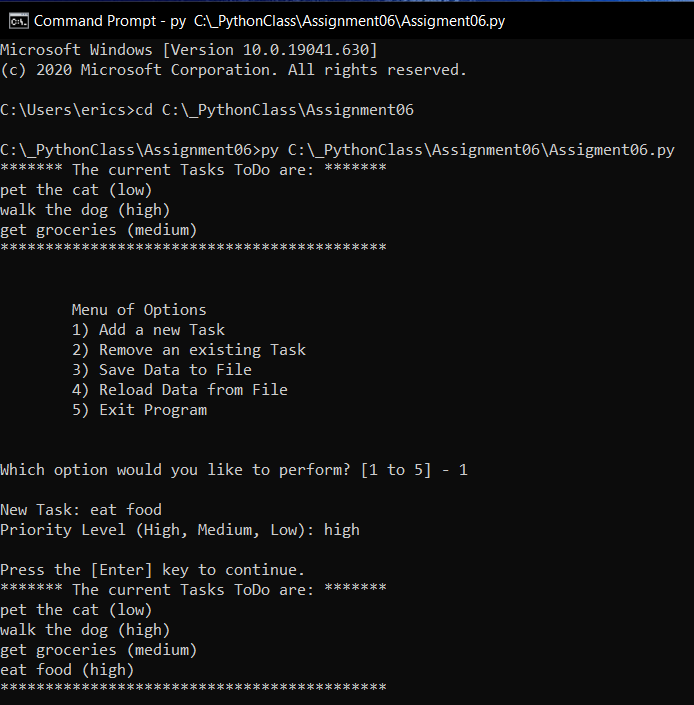
***Figure 8.4: Items saved to text file.***

When the user chose the number five, the program stopped.



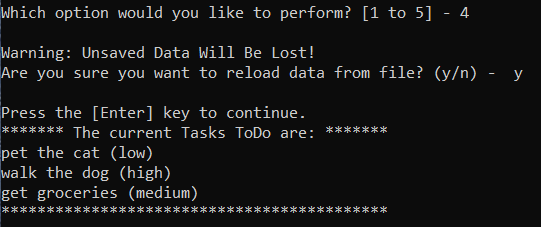
***Figure 8.5: User exiting program.***

Below shows the code ran through the command prompt, and the user adding one more item to our list.



***Figure 8.6: Code ran in the command prompt.***

When the user chose option four, you can see that the list reverted back to the original three items that were in the text file.



***Figure 8.7: Reloading data from text file.***

**Summary**

This assignment was another challenge where I had to modify someone else’s code. I found that the bulk of the assignment was fairly easy because I was able to re-use code from my previous ToDoList program. However, it was time consuming changing variables to match the original author’s variables. To speed this up I became familiar with ‘refactor’. Also, it was time consuming to bounce back and forth in the code from the function calls, to the function definitions. To solve this, I became familiar with ‘ctrl-B’ which allows you to navigate back and forth. Overall, I feel much more confident in static methods, return values, and program structure.