Justin Calhoun

Professor Summer Rae Elasady

Foundations Of Programming: Python

3 November 2018

Assignment 05 - ToDo List

This assignment was a bit odd for me, because I had a pretty good idea of how I wanted it to function from the beginning, and ended it up writing it almost entirely from start to finish, top to bottom, in one sitting. It was only tested once mid-creation to check output layout on the terminal, wasn’t run again until entirely complete, and only required a few corrected typos to function as intended.

I started it as I do most scripts, by wrapping it in a main() function, and then doing initial set-up. I knew I’d need a few variables up-front, so I created an empty ToDo list, a list of valid yes/no answers, a list of dictionary items for the menu, and a boolean flag for keeping track of when our ToDo list variable was out-of-sync with the ToDo.txt file.

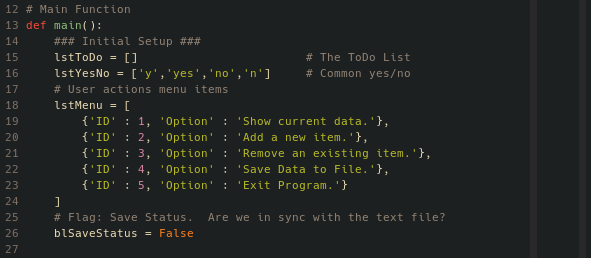


Figure 1: Initializing variables

Next, I used a try...except block to attempt to open the ToDo.txt file and read it’s contents into the ToDo list. If successful, the blSaveStatus flag is set to True. If the file doesn’t exist, I simply let the user know.

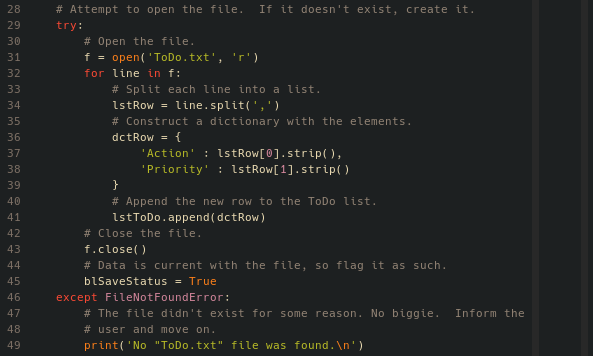


Figure 2: Reading the contents of ToDo.txt

With setup complete, I move into the main interactive part of the script. I first show the user whatever data we loaded from ToDo.txt (if anything).

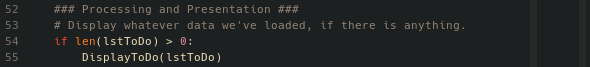


Figure 3: Initial data display

I knew that I’d be doing this at more than one place in the script, though, so I immediately created a function for it, DisplayToDo(). This function takes one parameter, the ToDo list variable, and parses it for nicely aligned output using the string format mini-language.

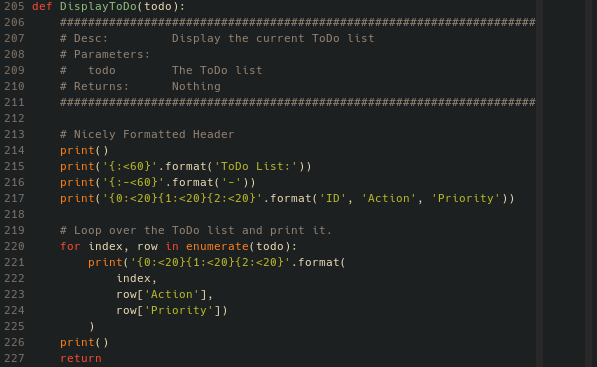


Figure 3: DisplayToDo() function

Now, I begin the main while loop that will persist until the user exits the program. On every loop, I present the menu to the user by simply looping over the list I created at the start. Then, I ask the user what they’d like to do, but I knew I’d want to validate their answers. Further, I knew I was going to be asking for a lot of input that I wanted to ensure was “valid”, so once again, I created a function to save myself the repetition.

The ValidateInput() function takes two parameters: the question we’re asking the user (as a string), and the answers we’ll accept from them (as a list). The function starts another while loop. It will ask the user for input by displaying the question and the accepted answers. Then, if the answer they provide is in the list, the function will return the value. If not, I inform the user to try again, and the loop repeats until they respond with something the script can work with.

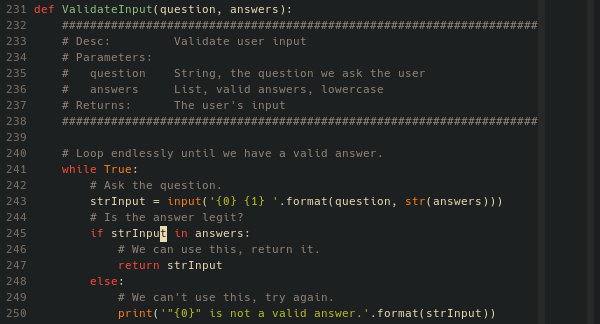


Figure 4: ValidateInput() function

With the function in place, I ask the user to select a menu item, and populate the answers list by building a quick list of the values associated with the ‘ID’ key in each of the dictionaries in the menu list variable.

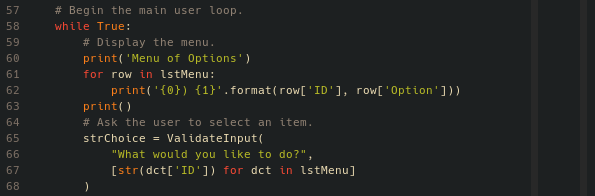


Figure 5: Printing the menu and asking for input

After this, I start an if...elif conditional with different actions based on each menu option. If they choose “1”, “Show current data”, I simply call DisplayToDo() again. If they choose “2”, “Add a new item”, I print a quick bit a feedback confirming they are about to add an item, ask them for the new action item (any string input will do, nothing to validate), and then ask them the priority using ValidateInput() and [‘low’,’med’,’high’] as valid answers. With user input collected, the values are packed into a dictionary, appended to the ToDo list variable, and I print a confirmation that the item was added. Finally, I set the blSaveStatus flag to False, as I know that the list in memory and on disk are no longer the same.

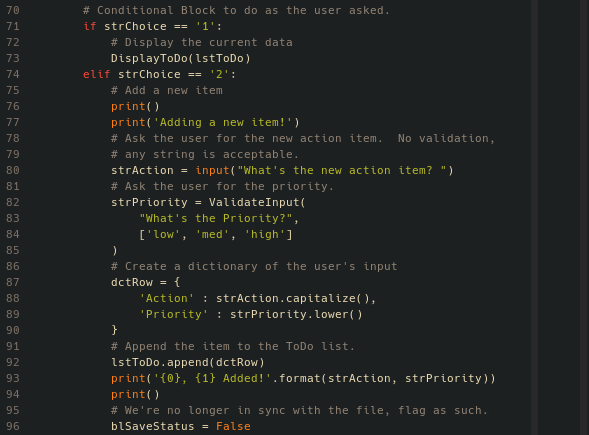


Figure 6: Menu options 1 and 2

If they choose “3”, “Remove an existing item”, I first build a list of indexes for the current items in the ToDo list. To that list, I add options to cancel, giving the user the opportunity to back out if they change their mind. With the answers list built, I use ValidateInput() to ask which item the user wants to delete. If they choose to cancel, I print a message confirming the cancellation, and that’s it for this loop.

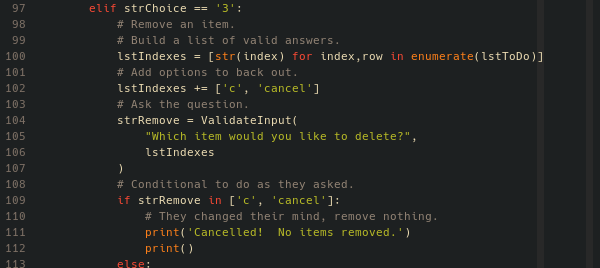


Figure 7: Menu option 3, part one

If they select a number, I print a message telling them exactly which item they are about to remove, and then ask them to confirm one last time. If they say “no”, they simply receive a message that the action was cancelled, but if “yes”, I delete the item at the given index from the ToDo list. Finally, if they did delete something, the blSaveStatus flag is set to False, as I once again know that the list in memory and the list on disk cannot be the same.

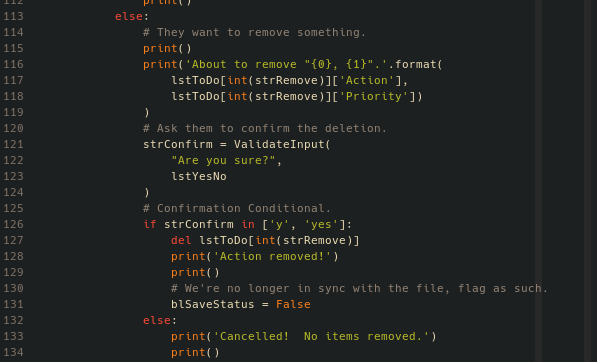


Figure 8: Menu option 3, part two

If they choose “4”, “Save Data to File”, I ask them if they really want to save. If they say “no”, I print a cancellation message. I knew that I’d want to reuse the code to save the file again in the final menu option, though, so I created my third and final function for this script.

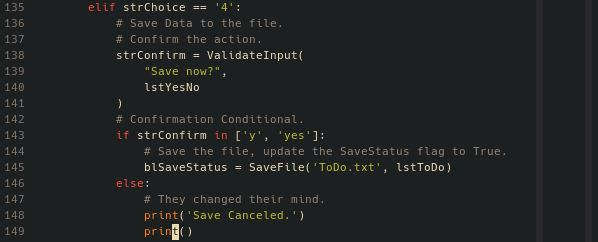


Figure 9: Menu option 4

The SaveFile() function takes two parameters: the file name to save to, as a string, and the ToDo list variable. It then opens the file with the given name, loops through the list to create strings, writes them to the file, and then closes the file. I print a quick confirmation that the file was saved, and then return True: this lets me also use the function’s return value to set blSaveStatus back to indicating the variable and file are in sync again.

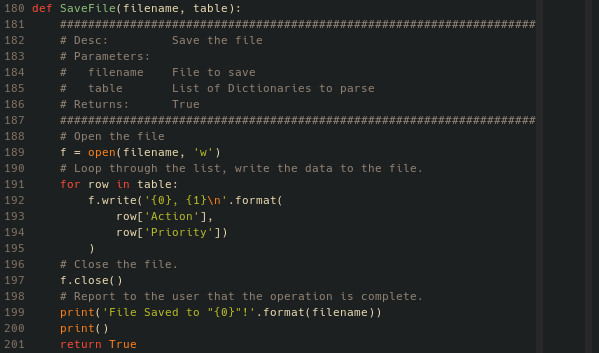


Figure 10: SaveFile() function

If they choose “5”, “Exit Program”, I ask them if they really want to exit. If they say “yes”, I check the current state of blSaveStatus, and if it’s False, I ask if they want to save. If “yes” again, I call SaveFile(). Regardless of whether the user chose to save, if they confirmed their choice to exit, I return from the main() function, effectively exiting the script.

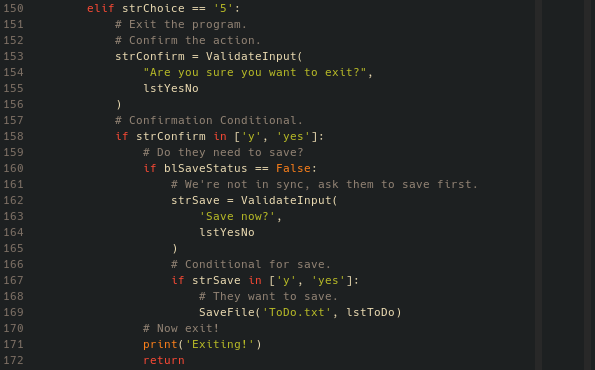


Figure 11: Menu option 5

Finally, and I’ll admit this is because I was feeling a bit cheeky, I added a superfluous else to the end of the conditional, but because the user cannot successfully input a value other than 1-5, the code inside can never run.

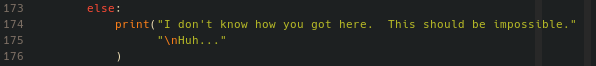


Figure 12: Print statement that never prints

As I stated at the beginning, other than needing to fix a few typos, all of the logic in this was done in one shot. It could stand to be a bit of a cleaner user experience, but in thinking about it, the best way for a user to modify the text file wouldn’t be to use a python script at all, but rather to open it in their favorite text editor. From that perspective, I’m happy with it as it is. Were I to continue improving it, though, I’d probably try to find good opportunities to clear the terminal, keeping clutter down. Going with that, I’d look into always display the current data after every screen clear, or at least some subset, with menu option 1 displaying the full contents. Finally, you’ll note in the below screenshot that the menu formatting can be a bit off with long Action items, so some tweaking to the column sizes, and some truncation on the strings would be necessary before I considered the polish complete.

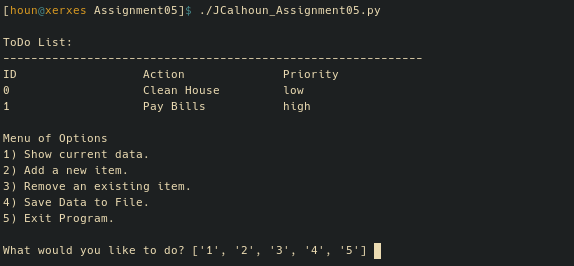


Figure 13: Script upon first open

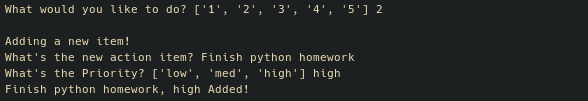


Figure 14: Adding an item

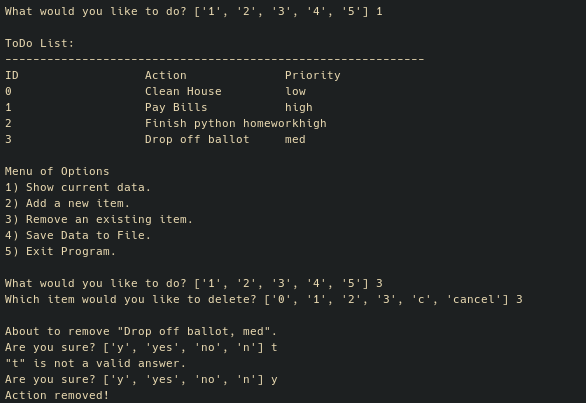


Figure 15: Removing an item

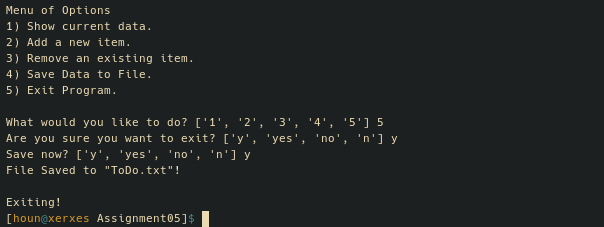


Figure 16: Exiting and saving