Meredith Shapiro

August 10, 2022

IT FDN 110 B - Foundations of Programming: Python

Assignment 06

Github: <https://github.com/meshapiro/IntroToProg-Python-Mod06>

Working with Functions and Classes

**INTRODUCTION**

In this paper I will explain how to write functions, by modifying last week’s homework to replace repetitive bits of code with functions instead.

This script accesses our familiar text file with a to-do list and performs the same actions as last time (use a menu to allow user to either add items, remove items, save and/or exit), but this time these actions will be performed using functions from a couple different classes – Processing and IO.

**WRITING THE SCRIPT**

**Declare Variables**

As in previous programs, I declare variables at the beginning

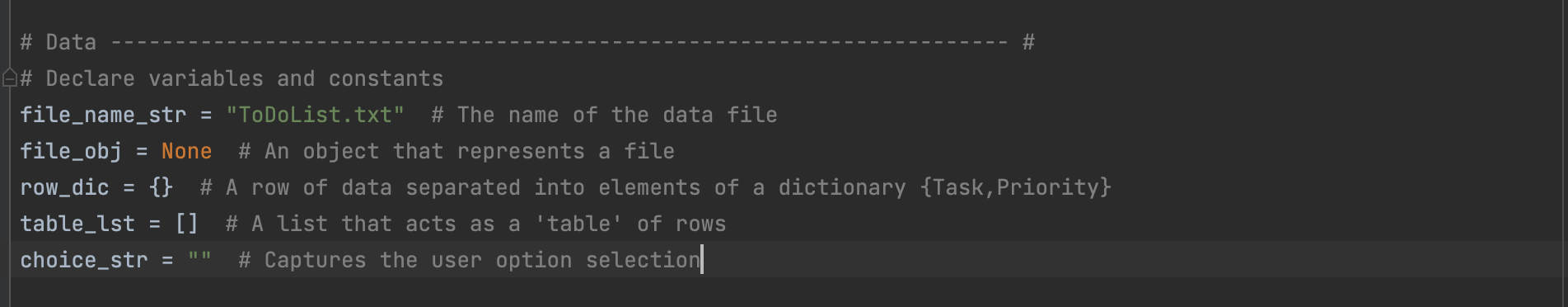


Figure 5. Variables declared

In Figure 5, you can see I ended up declaring fewer variables here than last week. This time, since some variables are created/used just within functions, I didn’t have as many to create up front.

**Processor Class**

A class is a way of grouping related functions. For this script, we use two classes, Processor and IO which are for exactly what it sounds like – one for reading/otherwise processing data, and one for input and output.

Within the Processor class, we create four functions:

**Read data**

First we need to read data. We will be doing the same thing as last week, within a function:

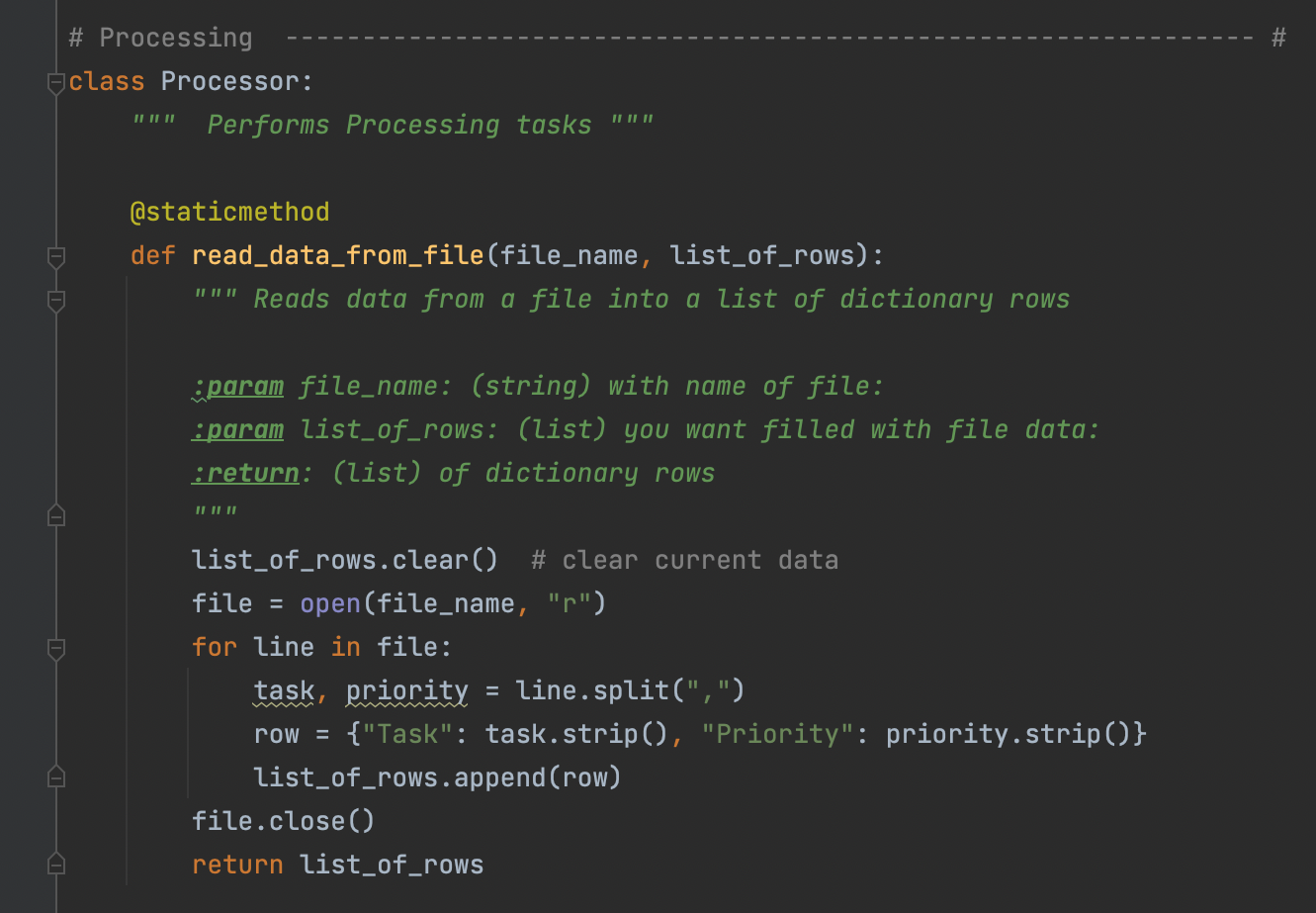


Figure 2. read\_data\_from\_file function

In Figure 2 the function read\_data\_from\_file is defined (“def” being a Python key word, meaning define). In the line “def read\_data\_from\_file(file\_name, list\_of\_rows):” we are telling Python that we are creating a function called read\_data\_from\_file, and that this function will be using two parameters, file\_name and list\_of\_rows. From there, the code is the same as we used last week, with the addition of “return list\_of\_rows” which will be the result from using that function. List\_of\_rows gets each new dictionary row appended to it.

**Add rows**

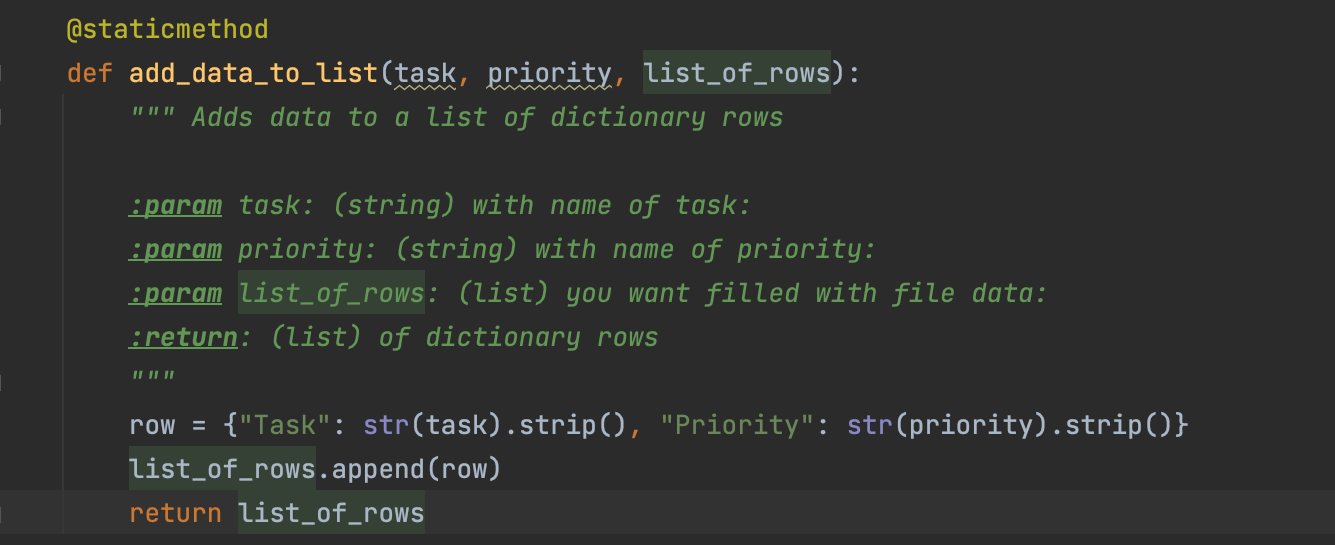
Next, we need a processing function that can add new data to an additional row. 

Figure 3. add\_data\_to\_list function

In Figure 3 you can see that the code used to add an entry to the to-do list is very simple. It will take a task and a priority as well as a list of rows, turn the task and priority into a dictionary row, then append that row to the existing list of rows and return the newly appended-to list at the end. **NOTE:** Since this is within the processor class, we don’t do anything here for gathering the user input for task and priority. This was slightly confusing to me at first, but now I appreciate the compartmentalized nature of the different functions.

**Delete rows**

Next is a function to remove a row. I used the exact same code as last week for this part, although I created the Boolean for row removed or not within the function rather than at the beginning of the script.

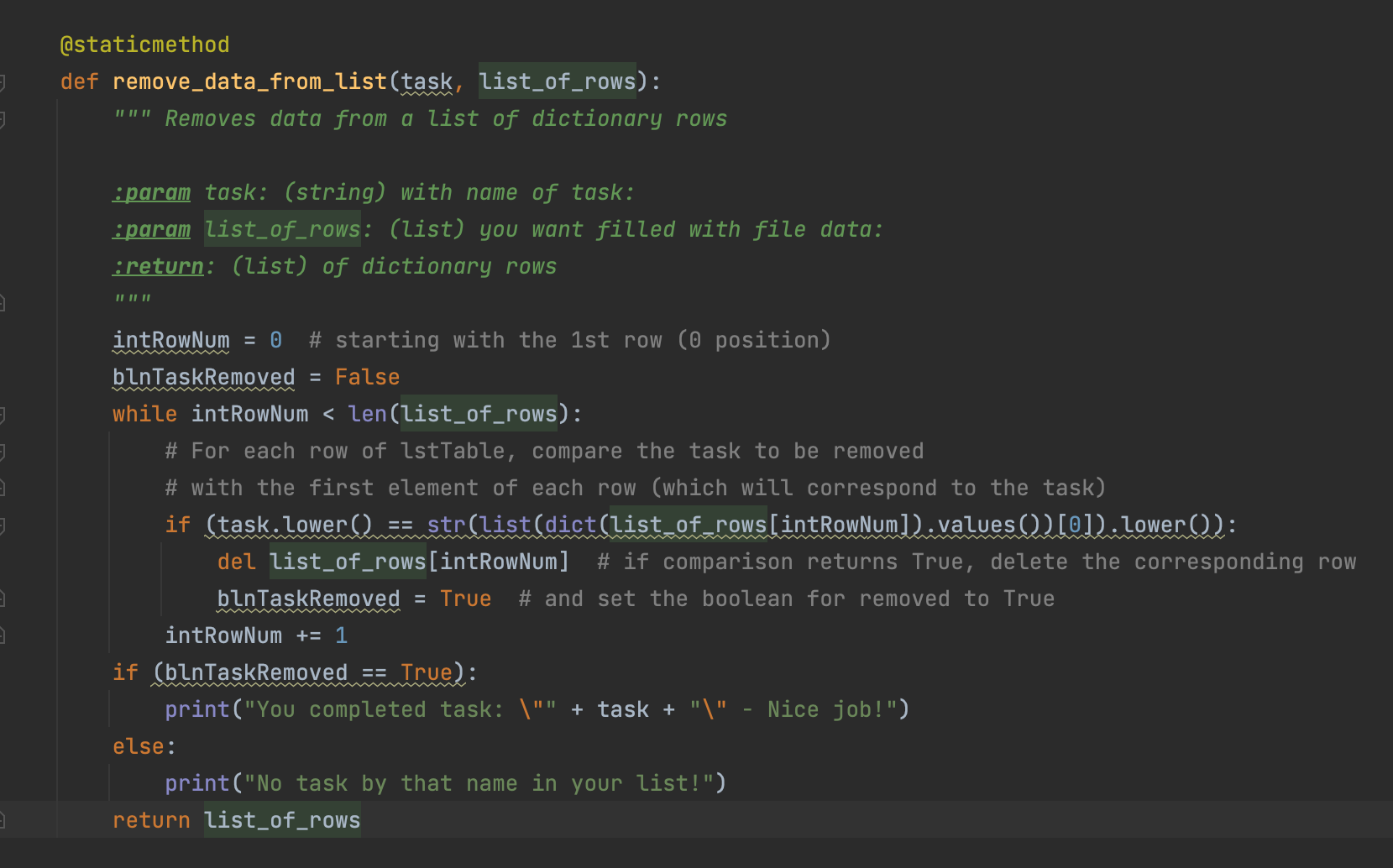


Figure 4. remove\_data\_from\_list function

I am now wondering if I did this right – for the “adding rows” function I only did the data processing, but as you see in Figure 4. I print messages regarding whether there was a match or not, and I wonder if those messages should be part of the IO section?

**Write Data**

Finally, we need a function for writing the data to the file. This is another block of code that is the same as I used last week, just wrapped up in a function now.

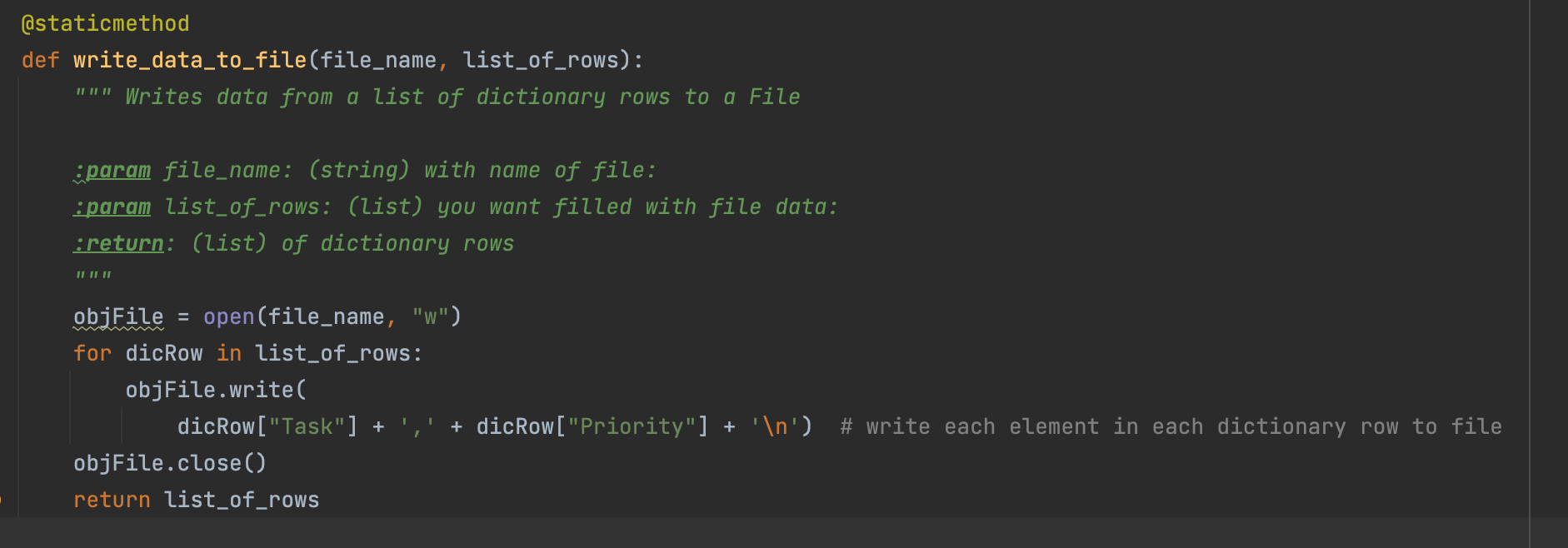


Figure 5. write\_data\_to\_file function

In Figure 5. You can see that the writing data code is the same as last week, with the addition of “return list of rows” statement. After writing, we still return the list of rows at the end so that if necessary we could print out what was just saved.

**IO Class**

Next we have a class for input/output of data. In these functions we will actually collect user input for which tasks they want to add or remove, and print out the items in the list each time. We also have functions for the display of the menu of options, and taking the user’s choice.

**Display menu, take user selection**

The first two functions are for the menu of options. First, one to print the options (output\_menu\_tasks) and one to take the user’s selection (input\_menu\_choice).

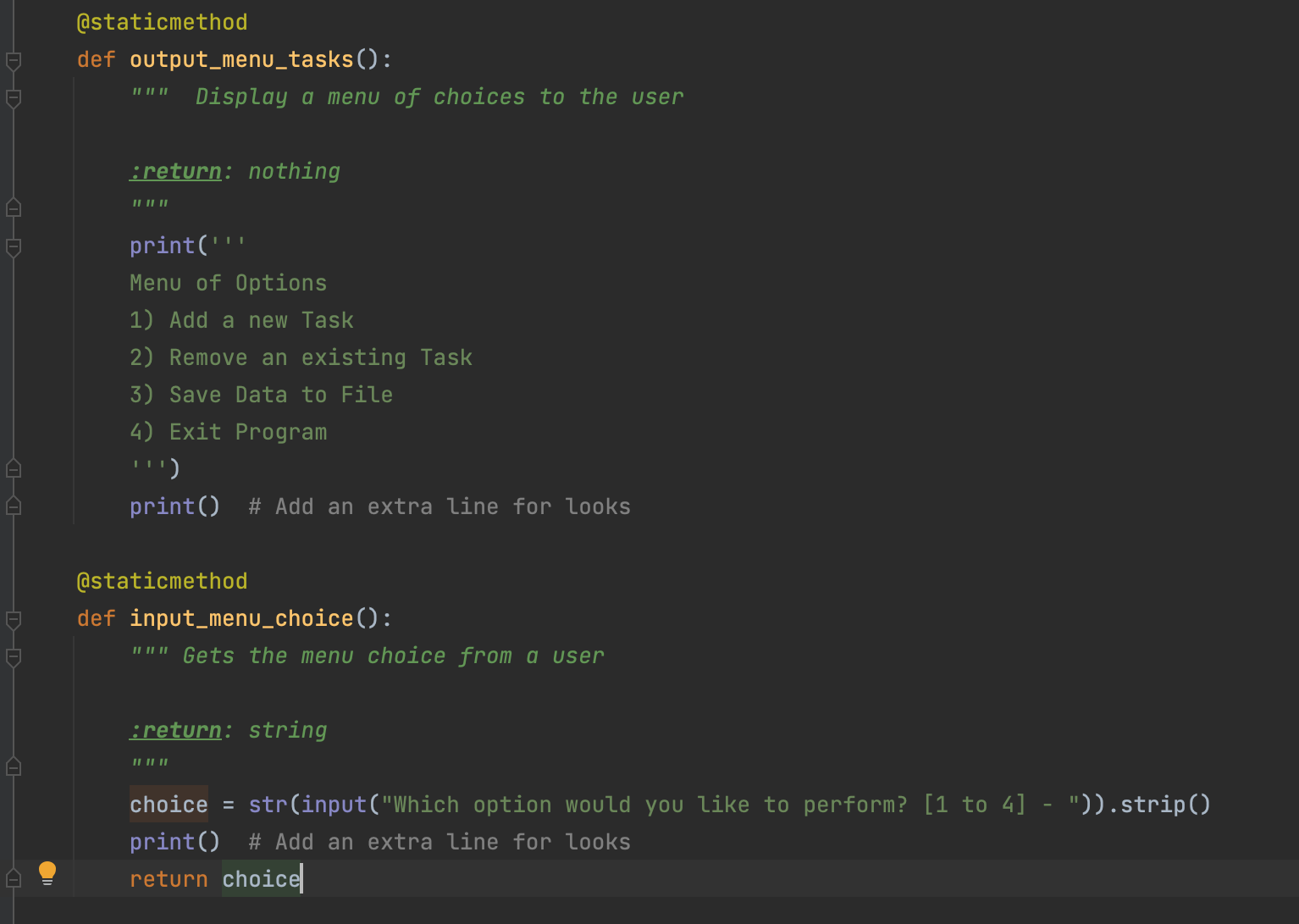


Figure 6. Two functions for menu options

In Figure 6 you can see the two different menu functions. Interestingly, the first function output\_menu\_tasks is really just for printing – so it doesn’t take any parameters and doesn’t return anything. It is just there to show the menu. For input\_menu\_choice, we have our first user input taken via function. We return the user choice value which will be used within the final loop.

**Print current data**

I next made a function to print all rows within the table – this code is so similar to last week’s that I am not showing it here, but the exciting part about this is that we can now use the function output\_current\_tasks\_in\_list() each time we want to print what is in our list, without repeating the code each time.

**Input new task and priority**

We next create a function to get user input for a new to-do list item.

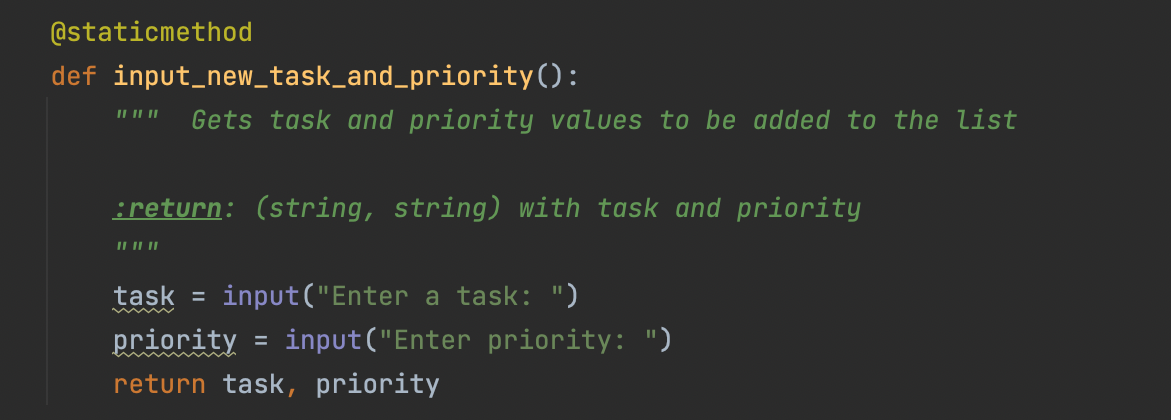


Figure 7. input\_new\_task\_and\_priority function

Figure 7 shows code used to get user input for an additional task and priority. In this section, all we do is gather the input and then return it – we will use the function in combination with Processor.add\_data\_to\_list to actually use the input.

**Input task to remove**

The function for gathering user input to remove a task is much like the input new task code – except even simpler since we just collect a task and not a priority.

**Main Body**

Now that we have our two classes and corresponding functions, the main script becomes very short, as it is just a While loop for comparing the user choice input against our if statements, and then within each IF we call the relevant functions.

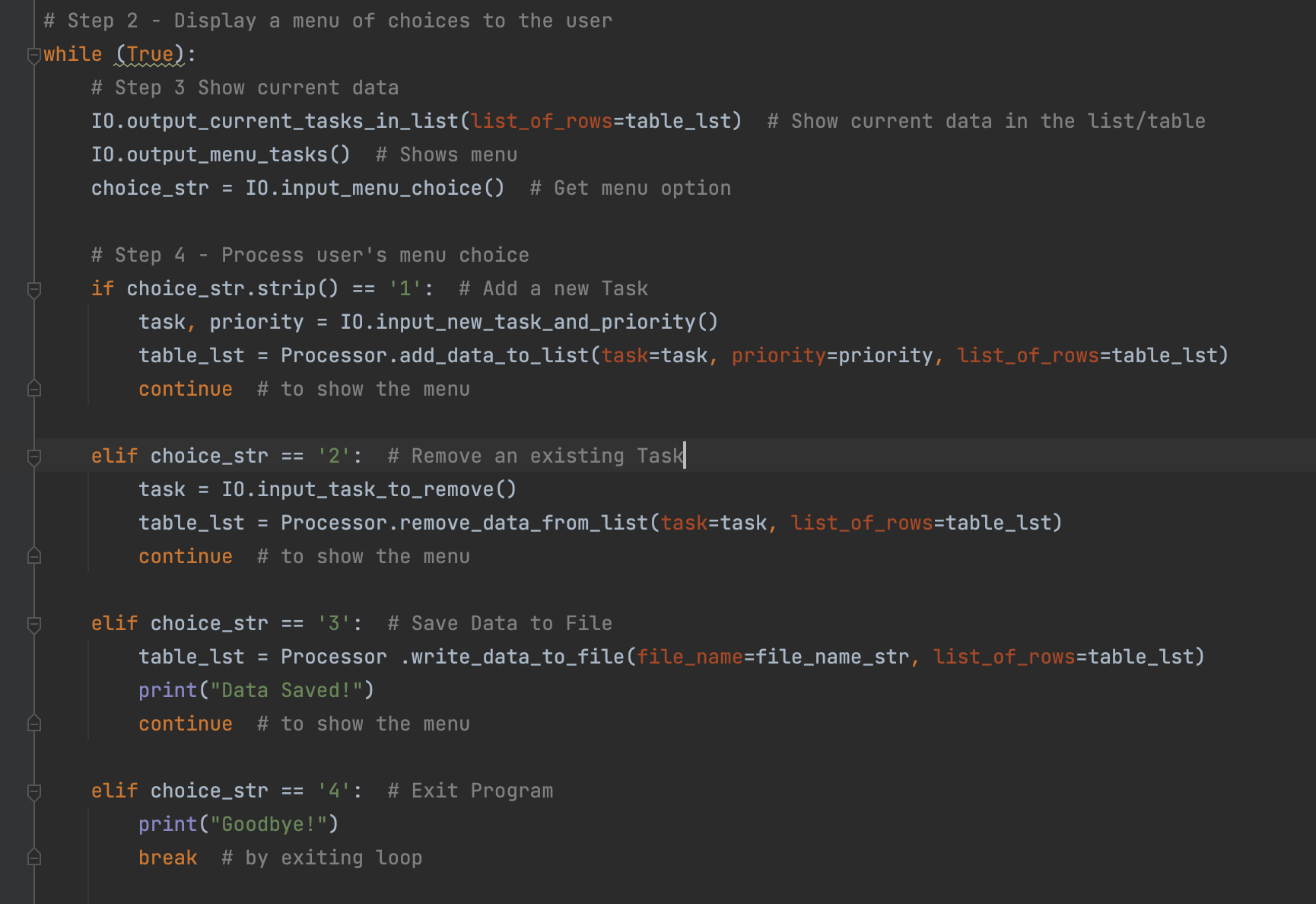
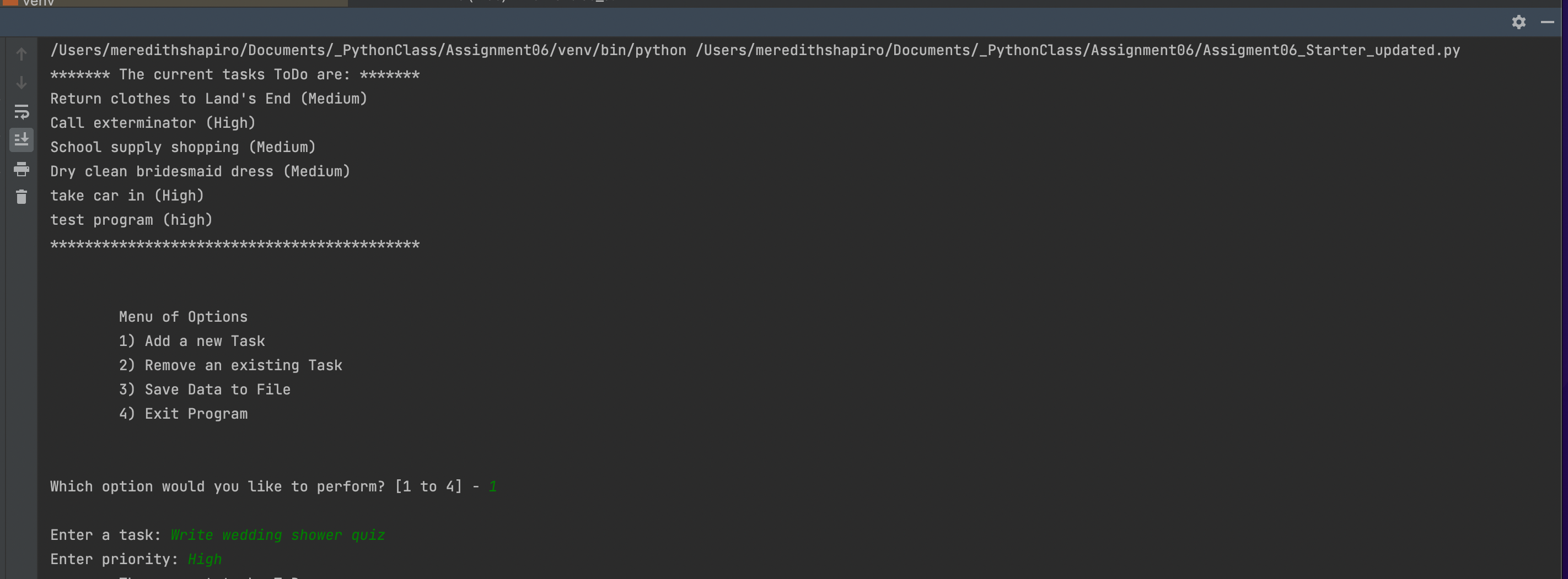


Figure 8. Main body of script, using our functions

Figure 8 shows how minimal the main script can become once we have all our functions. On the one hand, this is nice because it’s simple and it allows us to stop reusing redundant blocks of code, like the output\_current\_tasks\_in\_list function. On the other hand, if you don’t intimately recall what all happens in each of those functions, looking at a script that’s just a series of functions strung together can be somewhat confusing. It helps that we gave all the functions descriptive names that say what they are doing – that’s important for interpretability.

**RUNNING THE SCRIPT**

Here is how the script looks running in Pycharm:



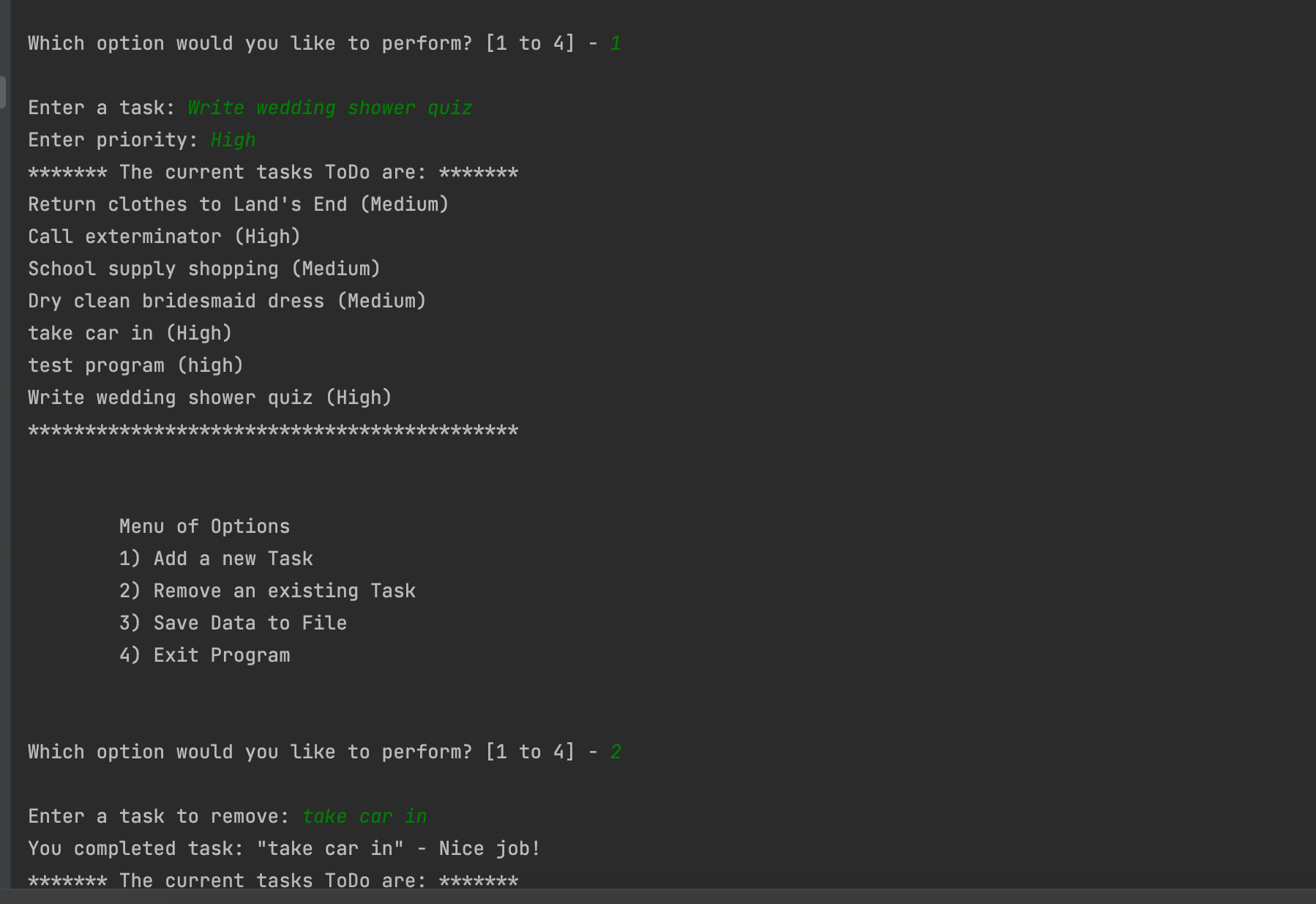


Figure 9. Script running in Pycharm

I did finally take my car in to have a tire replaced, so I crossed that off my to-do list. I need to finish writing a quiz for a wedding shower I’m hosting this weekend, so I added that.

I saved the file and can see that the text file got updated.



Figure 10. ToDoList.txt in Assignment folder

If I open it, I see the car task is gone and the wedding shower quiz task is there:

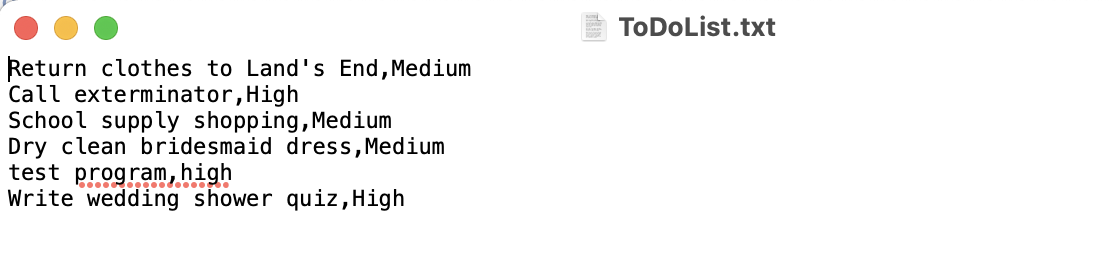


Figure 10. Updated text file

Next I will run it via the terminal, and remove the task for testing the program since we can see it is working.

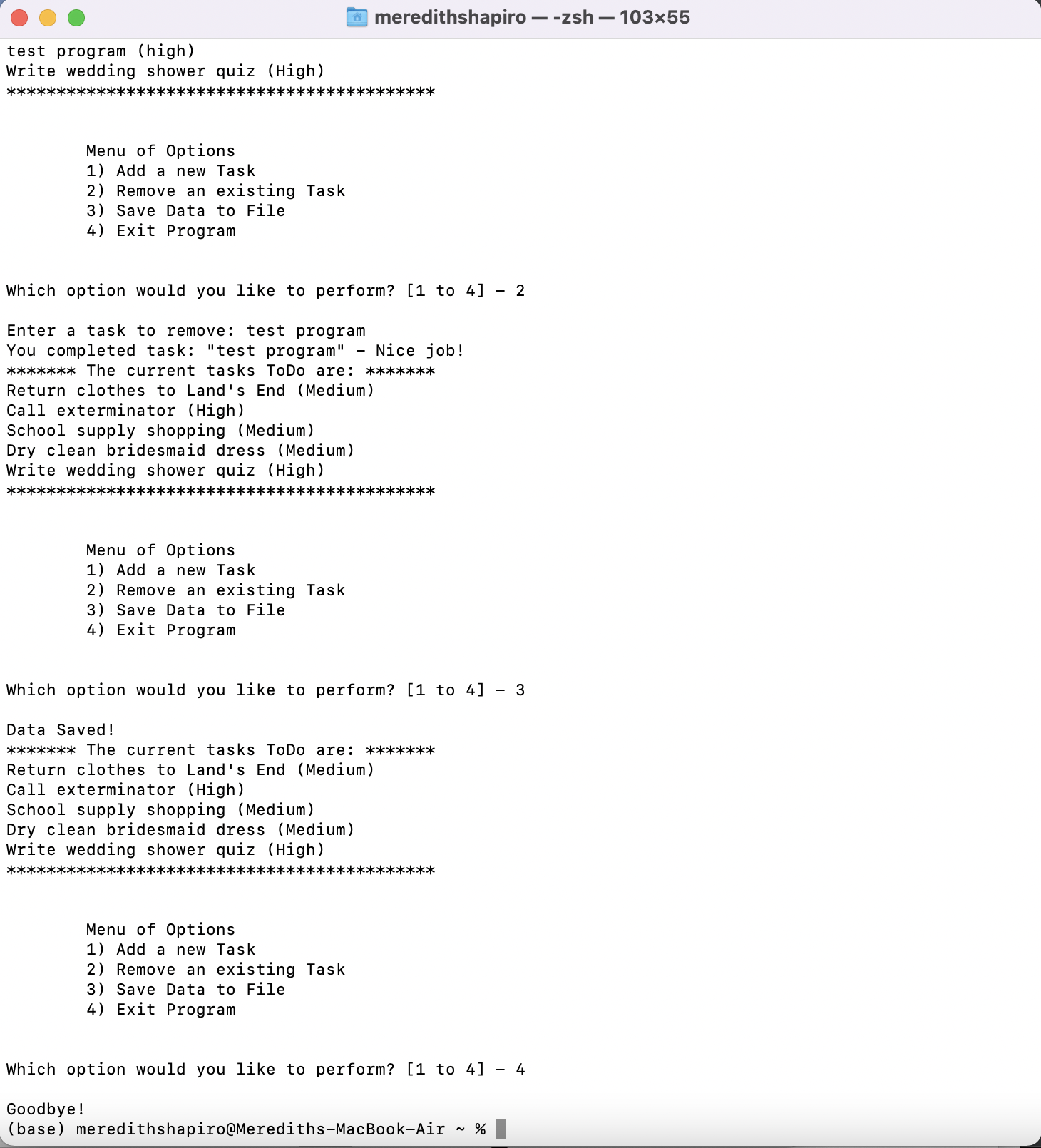


Figure 11. Program running via Terminal

In Figure 11 you can see that I removed one task and then saved again.

And confirm that task got removed:

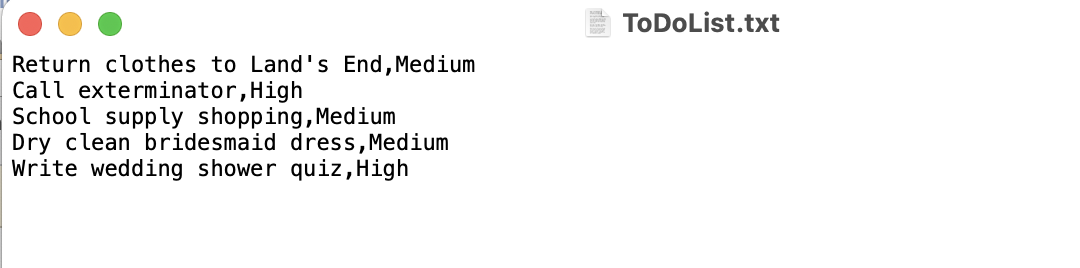


Figure 12. Latest update to ToDoList.txt

In Figure 12 you can see I did in fact remove the “test program” task.

**CONCLUSION**

In this module, we learned how using functions can further simplify our code and allow us to repeat redundant bits of code faster. I was intrigued to see how it can work to call multiple functions within the different parts of the loop, to first take input and then process it in separate steps.