Nathaniel Smith

May 25, 2020

Foundations of Programming: Python (IT FDN 100 A Sp 20)

Assignment06

https://github.com/10neg9/IntroToProg-Python-Mod06.git

Python: Script Using Functions and Classes

# Introduction

In this paper I discuss the Python code in the script called Assignment06.py. In this assignment I was provided with starter code Assigment06\_Starter.py, and it was my job to modify the code so that the script would run as desired.

The script reads and writes to the file ToDoList.txt. The data in the text file consists of a list of tasks and their corresponding priorities. The script allows the user to see this list and add to the list or remove items from the list. If desired, the user can throw out the changes by reloading the list from file. And when the user is done, the user has the option to save changes to the text file or exit without saving.

Although the script reads and writes to a file named ToDoList.txt, it uses dictionaries and lists to process the data in memory before it is written back to the file. This script makes use of while loops, for loops, if-elif-else statements, lists, dictionaries, functions and classes.

The above all sounds very similar to the last script, Assigment05\_Starter.py, but this time the script uses functions and classes for improved separation of concerns. Using functions and classes makes the code easier to edit, easier to debug, and easier to read. In this paper I will explain all the edits that I made to the original script to get it working.

# Script Header

As is good practice with any script, Assignment06.py begins with a script header, see lines 1 through 22 in Figure 1. The purpose of the script header is to describe what the script does and when it was created. And just as I have done, it’s also a good place to store the change log for the script. From the change log you can see all the edits that I made to the code to get it working. Throughout this paper I will discuss each of the changelog entries.

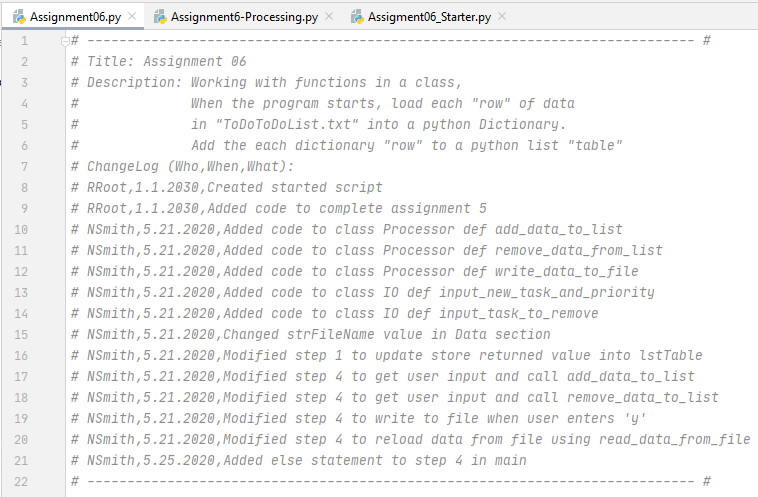


Figure 1. Assignment06.py Script Header

# Pseudocode

In this assignment the pseudocode was provided as shown below. Throughout the process of editing the code, I often looked at the pseudocode to keep myself reminded of what the code was trying to do.

*# Step 1 – When the program starts, Load data from ToDoList.txt*

*# Step 2 – Display a menu of choices to the user*

*# Step 3 – Show current data*

*# Step 4 – Process user’s menu choice*

*Option 1 – Add a task*

*Option 2 – Remove an existing task*

*Option 3 – Save Data to File (with option to cancel)*

*Option 4 – Reload Data from file (with option to cancel)*

*Option 5 – Exit Program*

# The Code

The code is too long to present in this paper. The code can be found in the file Assignment06.py which is stored in the same folder as this paper.

# Separations of Concern

The starter code was organized into three sections: data, processing, and presentation (input/output). This is a design principle known as separations of concern. In Assignment05-starter.py I tried to follow this principle, but I was not able to completely do so because it was difficult to separate some of the processing code from the input/output code. But in Assignment06.py, I was successful at implementing separations of concern and that is because I used classes: one class contains the processing functions and one class contains the input and output functions.

# Data Section: Declare Variables

All variables used in the script were declared in the data section. Strings are declared with a pair of quotation marks (either single or double quotation marks can be used), lists are declared with a pair of left and right square brackets, and dictionaries are declared with a pair of left and right curly brackets. The data section with comments is shown in Figure 2. I only made one change to the data section which was not even necessary. I changed the data file name, i.e. the string value, stored in the variable strFileName. I changed it from ToDoFile.txt to ToDoList.txt. This was a personal choice and it had no effect on the performance of the script.



Figure 2 Data Section

# Processing Section

The processing section consists of one class Processor. Inside Processor are four functions (but more precisely these are functions that have been converted to static methods using the @staticmethod decorator): read\_data\_from\_file(), add\_data\_to\_list(), remove\_data\_from\_list(), and write\_data\_to\_file().

# Processing Section: read\_data\_from\_file()

read\_data\_from\_file() is the only function in Processor that I did not edit, see Figure 3. The function takes two arguments: the name of a file and the name of a list. It returns a list of dictionary elements. This function is used to open a text file, read the text file line by line, and store each line in the text file as a dictionary element in a list. It returns a tuple containing the list list\_of\_rows and the string ‘Success’ to indicate, for debugging purposes I believe, that the function executed successfully.



Figure 3. Function read\_data\_from\_file()

# Processing Section: add\_data\_to\_list()

The function add\_data\_to\_list() is shown in Figure 4. The function has three parameters and as seen from the docstring this function takes two items (parameters 1 and 2), adds the items to a dictionary, and then it appends the dictionary element to a list (parameter 3).

The first thing I did was create the list taskvalues. Using a for loop, I copied all values of key “Task” of the dictionary elements in the list list\_of\_rows and stored them in the list taskvalues.

The if statement checks if the task passed to parameter 1 is in taskvalues. If it is, it means the user has already added this task to the to do list. In this case, the task is not created, instead the priority is updated to the argument passed to parameter 2 and the user is notified that the task already exists and the priority is updated. One improvement could be made here. I could have also checked if the priority matched the priority passed to parameter 2. If so, I could have had another message to indicate that no changes were made.

The else statement will run if the if statement evaluated to False. In other words, the else statement runs if the task passed to parameter 1 is not already in the list passed to parameter 3. If else executes, a dictionary element with a task and priority are appended to the list list\_of\_rows.

This function returns a tuple consisting of the list list\_of\_rows with changes (if applicable) and the string ‘Success’.



Figure 4.Function add\_data\_to\_list()

# Processing Section: remove\_data\_from\_list()

The remove\_data\_from\_list() function has two parameters: task and list\_of\_rows. The task parameter represents a task on the to do list that should be removed. The list\_of\_rows parameter represents the list that contains the up-to-date to do list.

Similar to the add\_data\_to\_list() function, the first thing I did was create the list taskvalues. Using a for loop, I copied all values of key “Task” of the dictionary elements in the list list\_of\_rows and stored them in the list taskvalues. Having done this twice, I could improve this script by creating another function that strips the task values and then call that function when needed.

The if statement checks if the task passed as an argument matches any task in the list taskvalues. If True, it iterates through each row in the list\_of\_rows. It checks if the task matches the task in the row, and if it does, the row is removed from the list\_of\_rows.

The else statement executes if the task was not found in the list taskvalues. And the user is notified with a message that the task was not found in the list and suggest to the user that maybe they made a typo.

This function returns a tuple containing the list list\_or\_rows with changes (if applicable) and the string ‘Success’.



Figure 5. Function remove\_data\_from\_list()

# Processing Section: write\_data\_to\_file()

The write\_data\_to\_file() function has two parameters: file\_name and list\_of\_rows. The file\_name parameter is used to open, write to, and close a file with the name of the value passed to file\_name. list\_of\_rows will contain the list of data to be written to file.

file\_name contains the value of the filename that is opened with write privileges. For each element in the list\_of\_rows, a task, a comma, the priority, and a carriage return are written to the file. The end result is a file with a to do list with each line representing a task and its priority. The file is then closed.

This function returns a tuple containing the list list\_of\_rows and the string ‘Success’.



Figure 6. Function write\_data\_to\_file()

# Presentation Section

The presentation section consists of one class IO. Inside IO are seven functions (but more precisely these are functions that have been converted to static methods using the @staticmethod decorator): print\_menu\_Tasks(), input\_menu\_choice(), print\_current\_Tasks\_in\_list(), input\_yes\_no\_choice(), input\_press\_to\_continue(), input\_new\_task\_and\_priority(), and input\_task\_to\_remove(). Of these seven functions, I only edited input\_new\_task\_and\_priority(), and input\_task\_to\_remove().

# Presentation Section: print\_menu\_Tasks()

I made no changes to the function in Figure 7. It has no parameters, it prints the menu options, and returns nothing.



Figure 7. Function print\_menu\_Tasks()

# Presentation Section: input\_menu\_choice()

I made no changes to the function in Figure 8. It has no parameters, it takes input from the user that is stored in the variable choice, prints a blank line for aesthetics, and it returns the string variable choice.



Figure 8. Function input\_menu\_choice()

# Presentation Section: print\_current\_Tasks\_in\_list()

I made no changes to the function in Figure 9. It has one parameter, list\_of\_rows. This function will print the to do list consisting of the values belonging to the keys “Task” and “Priority” in the list list\_of\_rows. It also prints a nice header and footer to separate the list from other text on the screen. It returns nothing.



Figure 9. Function print\_current\_Tasks\_in\_list()

# Presentation Section: input\_yes\_no\_choice()

I made no changes to the function in Figure 10. It has one parameter, message. The argument passed to the function will become text printed to the screen when prompting the user to make a yes or no choice. The function will return the user’s choice in lower case with all whitespace removed.



Figure 10. Function input\_yes\_no\_choice()

# Presentation Section: input\_press\_to\_continue()

I made no changes to the function in Figure 11. It has one parameter, optional\_message, that has a default argument, the empty string. This function will print the argument passed to it or a blank line if no argument is passed. Then it will ask the user to press the Enter key to continue. It returns nothing.



Figure 11. Function input\_press\_to\_continue()

# Presentation Section: input\_new\_task\_and\_priority()

The function in Figure 12 has no parameters. It asks the user to enter a task, stores the input to the variable task, asks the user to enter a priority, stores the input to the variable priority, and returns a tuple containing two elements, the variables task and priority.



Figure 12. Function input\_new\_task\_and\_priority()

# Presentation Section: input\_task\_to\_remove()

The function in Figure 13 has no parameters. It asks the user to input a task to be removed, stores the input to the variable task, and returns the variable task.



Figure 13. Function input\_task\_to\_remove()

# Main Body of Script

The main body contains all the pseudocode steps. I did not start working on this portion of the script until I had completed the code for the processing and presentation sections. I knew those sections were complete because I was testing them in another Python file, Assignment6-Processing.py. This file is stored with the rest of this assignment. I will not go into the details of that file here, but it is important to note that I did not start on the main body until I was confident in my code after thorough testing of the processing and presentation code.

# Main Body: Step 1

Step 1 of the main body, shown in Figure 14, is a nice and clean one liner. I had to edit this line so that the return of the function call would be stored in the variable lstTable. Also, I had to index the return (note the [0] at the end of the statement) so that only the list was stored in the variable lstTable. Remember that the read\_data\_from\_file() function returns a list and the string ‘Success’ i.e. it returns a tuple, but I need only the list in index position 0.



Figure 14. Main Body Step 1

# Main Body: Step 2 and 3

Steps 2 and 3 of the main body, shown in Figure 15, are part of an infinite while loop that shows the current to do list, then shows the menu of options, then asks the user to select a menu option, and stores the user selection to the string variable strChoice. This is performed in four lines of code and 3 of these lines are function calls to functions described above. I did not make any edits to this portion of code.



Figure 15. Main Body Step 2 and 3

# Main Body: Step 4

Step 4 of the main body, shown in Figure 16, processes the user’s input at the menu screen and performs 1 of 5 operations based on the input from the user. It is structured as an if-elif-elif-elif-elif-else statement.

The if statement checks if the user entered ‘1’. If True, the function add\_data\_to\_list() of the Processor class is called. Two arguments are passed in the form of inputs from the user (a task and a priority), and the return list is stored in lstTable. Then the input\_press\_to\_continue() function of IO class is called and the script holds until the user presses the enter key. Once the user presses enter, the continue statement is executed and the while loop runs again.

The first elif statement checks if the user entered ‘2’. If True, the function remove\_data\_from\_list() of the Processor class is called. Two arguments are passed. The first argument is input from the user representing a task to be removed and the second argument is the list that contains the to do list. The return list is stored in lstTable. Then the input\_press\_to\_continue() function of IO class is called and the script holds until the user presses the enter key. Once the user presses enter, the continue statement is executed and the while loop runs again.

The second elif statement checks if the user entered ‘3’. If True, the block of code first calls the input\_yes\_no\_choice() function of the IO class and passes the argument, "Save this data to file? (y/n) - ". The return string is assigned to strChoice. Then an if-else statement checks if the strChoice in lowercase is equal to “y”. If so, the write\_data\_to\_file() function of the Processor class is called and the arguments strFileName and lstTable are passed. The result is that the contents of the list lstTable are written to file. Then the input\_press\_to\_continue() function of IO class is called and the script holds until the user presses the enter key. If the if statement did not evaluate to True (i.e. the user did not enter “y” or “Y”), then nothing is saved to file and the input\_press\_to\_continue() function of IO class is called with the argument “Save Cancelled!”. The message “Save Cancelled!” is printed to screen and the script holds until the user presses the enter key. Once the user presses enter, the continue statement is executed and the while loop runs again.

The third elif statement checks if the user entered ‘4’. If True, a warning message, “Warning: Unsaved Data Will Be Lost!”, is printed to screen and the input\_yes\_no\_choice() function of the IO class is called with the argument “Are you sure you want to reload data from file? (y/n) – “. The argument is printed to screen and the input from the user is assigned to the variables strChoice. Then an if-else statement checks if the strChoice in lowercase is equal to “y”. If so, the read\_data\_from\_file() function of the Processor class is called with the arguments “ToDoList.txt” and lstTable. The function returns a tuple with the list in index position 0, which is assigned to variable lstTable which results in the list lstTable being reloaded with the data from file. Then the input\_press\_to\_continue() function of IO class is called and the script holds until the user presses the enter key. If the user did not enter “y” or “Y” the else statement will execute which calls the input\_press\_to\_continue() function of IO class with the argument “File Reload Cancelled!”. The message “File Reload Cancelled!” is printed to screen and the script holds until the user presses the enter key. Once the user presses enter, the continue statement is executed and the while loop runs again.

The fourth and final elif statement checks if the user entered ‘5’. If True, the message “Goodbye!” is printed to screen and the break statement executes to end the while loop and in turn ends the script.

The else statement executes if neither 1, 2, 3, 4, nor 5 are entered at the menu selection screen. A message notifies the user what was input and that the input is not a valid selection. Then the input\_press\_to\_continue() function of IO class is called and the script holds until the user presses the enter key. Once the user presses enter, the while loop has completed a loop and runs again.



Figure 16. Main Body Step 4

# Testing the Script With PyCharm

I first tested the script using PyCharm. You can see the script running in Figure 17 to Figure 22 below.

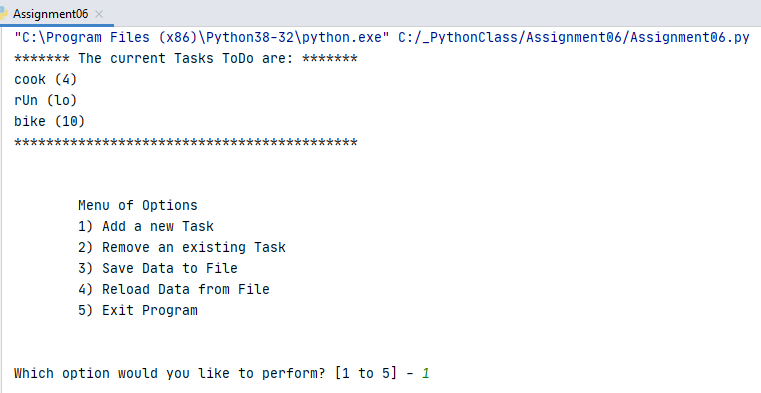


Figure 17. Assignment06.py Startup in PyCharm

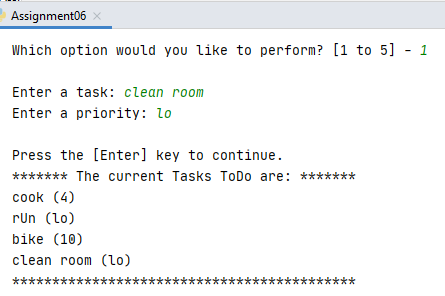


Figure 18. Add an item

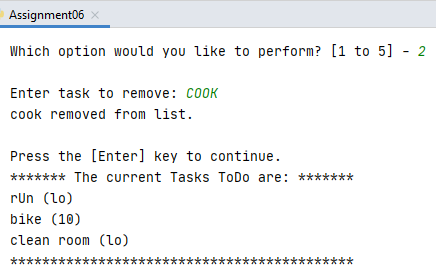


Figure 19. Remove an Item

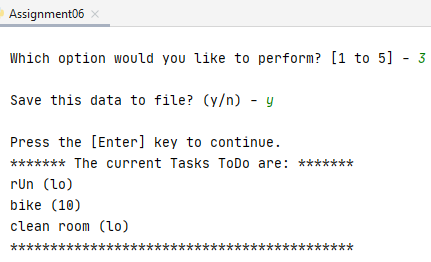


Figure 20. Save Data to File

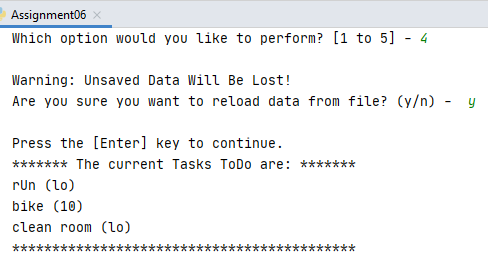


Figure 21. Reload Data from File

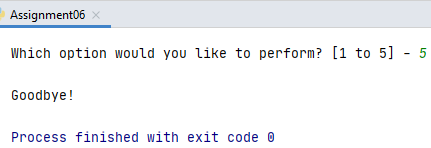


Figure 22. Exit Program

I checked the text file after running the code in PyCharm and it did work. The task ‘clean room’ with priority ‘lo’ was added to the ToDoList.txt, see Figure 23.

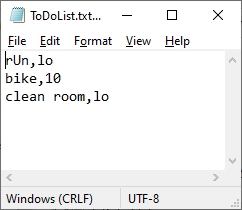


Figure 23. ToDoList.txt After Running Script in PyCharm

# Testing the Script With Windows Command Prompt

Now that I had it running in PyCharm, I also tested my script from Windows Command Prompt as shown in Figure 24 to Figure 29.

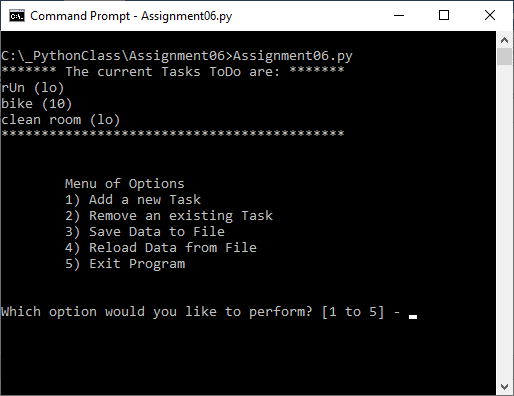


Figure 24. Running Assignment06.py in Windows Command Prompt

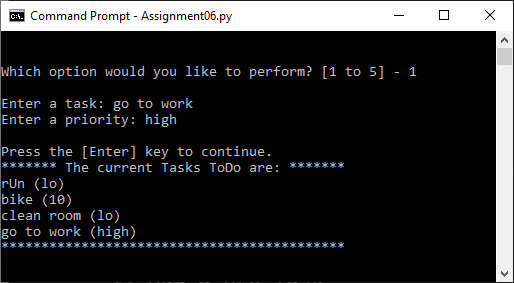


Figure 25. Add an Item

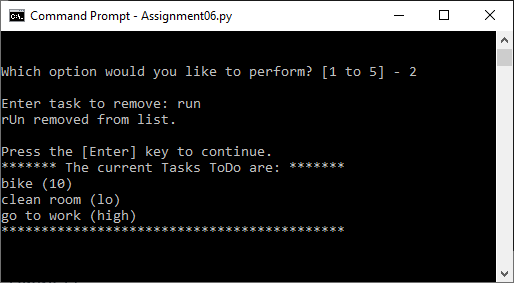


Figure 26. Remove an Item

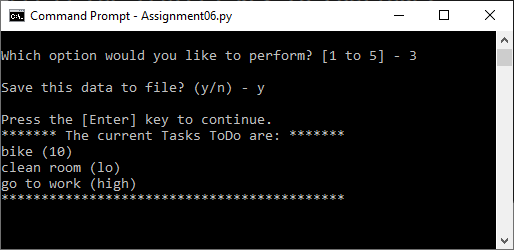


Figure 27. Save Data to File

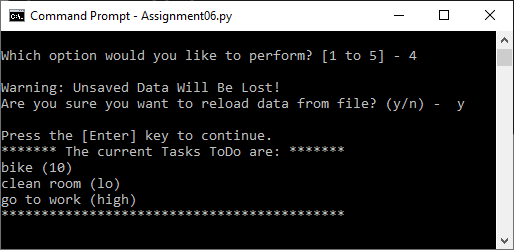


Figure 28. Reload Data from File

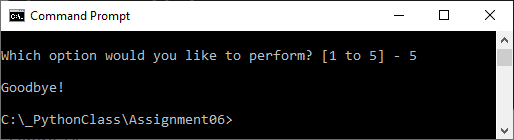


Figure 29. Exit Program

After running the script in Command Prompt, I ended up with the text file shown in Figure 30. The task ‘run’ was removed and the task ‘go to work’ was added. Complete success!

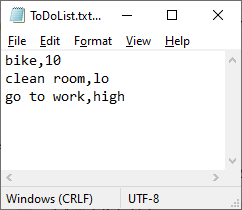


Figure 30. ToDoList.txt After Running Script in Command Prompt

# Summary

In summary, it was a bit daunting at first sight to have to edit 178 lines of code, but in the end, I was successful at getting the code to work. Because the code was separated into sections, it made editing the code much more manageable. It’s clear to me that classes and functions are helpful in reducing the amount of code that you have to write and they help to reduce the amount of errors in your code. Another benefit is that I was able to isolate and test each function on its own instead of trying to test an entire script which helped to speed up the debugging process.