Jessi Velazquez

11-22-2020

Foundations of Programming: Python

Assignment 06

<https://github.com/JessiVelazquez/IntroToProg-Python>

**Task Manager Program, Upgraded**

**Introduction**

This document summarizes the Python code behind a program used to manage a user’s constantly changing To-Do list. The program allows the user to add or remove items from their to do list and save the list at any time to a text file. The program also contains a function to re-load the data from the most recently saved version of the text file.

**Functionality of the Script**

This script is broken into three categories: processing code, presentation/IO (IO = input/output) code, and the main body code. We do this to maintain Separation of Concerns (SOC), or the ability to remove our code regarding data processing and presentation/IO from the main body of the script for ease of using blocks of code for different programs, or to isolate blocks of code for debugging, or to plug blocks of code from other programs into the current program.

In this program, we achieve SOC solely through the use of functions. Functions allow us to define processes or tasks through blocks of code using only parameters, which at first contain no value (or at times, a default value). This can be thought of like math functions such as X + Y = Z, where X, Y have not yet been given numeric values. The function is merely a skeleton or framework which can be run numerous times with different values assigned to X and Y. But the key here is that X + Y = Z as a function only exists in one place, and we merely re-assign, or “pass” our values to the parameters X and Y.

In Python the term for using the function, or assigning values to parameters is “calling” the function. This is what we do in the main body of our code. By merely calling functions in the main body of the code, it is much easier to follow along with what is happening because there is less code – all of the details of what is happening within a function is located elsewhere in the the processing of presentation/IO code blocks, and all we see is a nice, succinct and self-explanatory function call such as “remove\_data\_from\_list”.

With this in mind, this document will narrate the process of the main body of code, and refer to functions in detail when needed.

The first thing this script does is it calls the function Processor.read\_data\_from\_file. This calls a code block from the Processor “class” of functions that clears the current to-do list stored in memory, and replaces it with a new list it gets from splitting the data in ToDoFile.txt by comma, creating a dictionary item from each row, and appending this rows to the master list. Now that we have the data stored in memory, the program closes its connection to the text file.

Moving along, we enter a while(True) loop and display to the user the current list read in from the text file along with a menu of options and a prompt to select an option. We then enter an “if/elif” sequence. If the user selects the first option, it means he or she would like to add a task to the list. We take their input as the two variables in the function IO.input\_new\_task\_and\_priority, pack these in tuple form, and then unpack that tuple to the variables strTask and strPriority in the main body of code. We then pass these user input variables into the parameters task and priority within the function Processor.add\_data\_to\_list, which in turn assigns the input variables as the value components of a dictionary item, corresponding to the keys “Task” and “Priority”, which is then appended to the master data list table. Finally, we ask the user to hit the enter key to continue back to the main menu. See main body and processing code below:

Main body:

if strChoice.strip() == '1': # Add a new Task  
 strTask, strPriority = IO.input\_new\_task\_and\_priority()  
 Processor.add\_data\_to\_list(strTask, strPriority, lstTable)  
 IO.input\_press\_to\_continue(strStatus)  
 continue # to show the menu

Processor.add\_data\_to\_list:

@staticmethod  
def add\_data\_to\_list(task, priority, list\_of\_rows):  
 *""" Adds a row of data to Task List data table* ***:param*** *task: (string) Name of a task:* ***:param*** *priority: (string) Priority assigned to task (high/med/low):* ***:param*** *list\_of\_rows: (list) Task list data table* ***:return****: (list) of dictionary rows  
 """* task = strTask  
 priority = strPriority  
 row = {"Task": task, "Priority": priority}  
 list\_of\_rows.append(row)  
 return list\_of\_rows, 'Success'

Option 2 is to remove an item from the list. This is similar, however we never deal with information regarding priority under this option, instead opting to ask the user for the name of the task to remove, and are able to remove both that task and its priority since these items exist together as a dictionary row. We collect the user’s input by calling the function IO.input\_item\_to\_remove, assign that to global variable strTask, and then pass strTask into the function Processor.remove\_data\_from\_list. This function uses a for loop to look at the “task” dictionary key (similar to a column) in each row of data for a value matching the user’s input, and when it finds one it removes that row. But there is more. When this function is first called, we establish a local variable “count”, equal to 0. Under the if statement nested within the for loop of this function is a line of code that tells the program to increase the value of “count” by one IF a row is removed. Under this logic, if the program finds no values associated with the “task” key in the master list, and no rows are removed, the value of count will remain at 0 after the for loop is finished. Thus, after exited the for loop (but still within the function), we have another if statement that says if count is equal to 0, call the function IO.user\_message, and we pass the string “item not found, nothing to remove” into its parameter, which displays this message to the user. In short, if the user attempts to remove an item that does not exist in the list, the program will notify he or she of that.

Here is the main body code:

elif strChoice == '2': # Remove an existing Task  
 strTask = IO.input\_task\_to\_remove()  
 Processor.remove\_data\_from\_list(strTask, lstTable)  
 IO.input\_press\_to\_continue(strStatus)  
 continue # to show the menu

Here is the function Processor.remove\_data\_from\_list:

@staticmethodt  
def remove\_data\_from\_list(task, list\_of\_rows):  
 *""" Removes a row of data from the Task list data table* ***:param*** *task: (string) Name of Task to Remove* ***:param*** *list\_of\_rows: (list) Task list data table* ***:return****: (list) of dictionary rows  
 """* task = strTask  
 count = 0  
 for row in list\_of\_rows:  
 if task in row["Task"]:  
 list\_of\_rows.remove(row)  
 count += 1  
 if count == 0:  
 IO.print\_user\_message("Task not found in list, nothing to remove.")  
 return list\_of\_rows, 'Success'

Option 3 is to save the current data to file. If the user selects this option, we assign the global variable strChoice a value based on the user’s input from function IO.input\_yes\_no\_choice. If that choice is “y”, we pass global variables objFile and lstTable into the function Processor.write\_data\_to\_file, as its two parameters file\_name and list\_of\_rows.

*------------------------------------------------------------------*

*As we mentioned in our intro, it is important to remember that the function parameters “file\_name” and “list\_of\_rows” are like X and Y in X + Y = Z. The global variables we pass into them (objFile and lstTable) are like using numbers in place of X and Y, like 3 + 5 = 8.*

*--------------------------------------------------------------------*

This function then opens ToDoFile.txt in write mode, and using concatenation writes the variable (not the key) of each item of row in list\_of\_rows (*remember that is lstTable because we passed lstTable to list\_of\_rows*) as a new line in the file, separated by comma. So, the keys are “Task” and “Priority”, but the variables are user inputs like “Wash Car”, “Low”, and “Pay Bills”, “High”. So, we get a nice comma separated text file as our output. Finally, the program closes its connection to the file. The code for the Processor.write\_data\_to\_file function is below:

@staticmethod  
def write\_data\_to\_file(file\_name, list\_of\_rows):  
 *"""* ***:param*** *file\_name: Text file to be opened and written to by program.* ***:param*** *list\_of\_rows: (list) Task list data table* ***:return****: (list) of dictionary tows  
 """* # TODO: Add Code Here!  
 file\_name = open("ToDoFile.txt", "w")  
 for row in list\_of\_rows:  
 file\_name.write(row["Task"] + "," + row["Priority"] + "\n")  
 file\_name.close()  
 return list\_of\_rows, 'Success'

Back to the main body of code, we then call the functions IO.user\_message (passing the string “File saved!” to its optional\_message parameter), and IO.input\_press\_to\_continue, to first notify the user of a successful save, and then prompt them to press the enter key to continue back to the main menu.

Importantly, before we exit this code block within our main body, we are still within its if/else statement, and the “else” condition arises. Back at the start of option 3, we prompted the user input “y” or “n”, and went through all of the code that runs if the input is “y”. If the input was “n”, we simply call the IO.input\_press\_to\_continue to prompt them to press the enter key to return to menu. Below is the main body code for option 3:

elif strChoice == '3': # Save Data to File  
 strChoice = IO.input\_yes\_no\_choice("Save this data to file? (y/n) - ")  
 if strChoice.lower() == "y":  
 Processor.write\_data\_to\_file(objFile, lstTable)  
 IO.print\_user\_message("File Saved!")  
 IO.input\_press\_to\_continue(strStatus)  
 else:  
 IO.input\_press\_to\_continue("Save Cancelled!")  
 continue # to show the menu

Next is option 4, reload data from file. This essentially calls the same function that the program opens with, Processor.read\_data\_from\_file, though this time “voiced” as the task of restoring data (lstTable) to its previously saved version. This highlights the difference between data in memory and data saved. When we open the program and read in a ToDoFile.txt file with say, 4 tasks in it, those tasks go to memory for us to work with inside of our program. The connection to ToDoList.txt is closed. So, we could add 2 tasks and delete one task, but those changes only exist in memory unless we save them to file (option 3). If we close the program without saving, we are left with the same ToDoFile.txt we started with. This is “losing unsaved information”, a risk we are all familiar with, at least in Windows style applications.

Enter option 4. This option allows the user to revert intentionally, for whatever reason they may desire. Maybe they added some of their partner’s tasks to their own list and want to undo that work without closing the program. To accomplish this for them, we call function IO.print\_user\_message, passing the string “Warning: Unsaved data will be lost. Are you sure you want to continue?” Remember, the print\_user\_message function is just a simply function to print one statement. We reuse this function many times in the main body code by simply passing different string values, or text we want to display, to its one parameter “optional\_message”.

After displaying that message, we give the user a y/n option by calling function IO.input\_yes\_no\_choice and assigning its return value (either a y or n from the user) to global variable strChoice. We then enter an if/else statement where if strChoice is y, we call the same function we started our program with, Processor.read\_data\_from\_file, and send the user a “success” message with another instance of IO.print\_user\_message.

If the user selected “n” after the original warning message, they get prompted to hit enter to continue to main message, but with the message “File reload cancelled!” which is passed as a string into the parameter of function IO.input\_press\_to\_continue.

Here is the main body code:

elif strChoice == '4': # Reload Data from File  
 IO.print\_user\_message("Warning: Unsaved data will be lost!")  
 strChoice = IO.input\_yes\_no\_choice("Are you sure you want to reload data from file? (y/n) - ")  
 if strChoice.lower() == 'y':  
 Processor.read\_data\_from\_file(strFileName, lstTable)  
 IO.print\_user\_message("File reloaded. Data restored to last saved version.")  
 IO.input\_press\_to\_continue(strStatus)  
 else:  
 IO.input\_press\_to\_continue("File Reload Cancelled!")  
 continue # to show the menu

If the user selects option 5, they want out. So, we simply call and instance of IO.print\_user\_message, passing the string “Hasta Luego!”, and then use the break statement, exited the while(True) loop in which this entire program resides.

**Summary**

In this document, we have described the process that our script undergoes to perform actions on a user’s To-Do list managed via a text file of comma separated values. The user is given control over adding, removing, saving the list, and reloading the list. The elements that are key to this document are the explanations of how we use the Separation of Concerns (SOC) programming principle via defining and then calling functions throughout our script. This cleans up our script in many ways.