Ryan Lupinski

02.18.2022

Foundations of Programming: Python

Assignment 06

To Do List Improved

[1 Introduction 1](#_Toc96159534)

[2 Functions 1](#_Toc96159535)

[3 Classes 2](#_Toc96159536)

[3.1 Organization 2](#_Toc96159537)

[3.2 Separation of concerns 3](#_Toc96159538)

[4 To Do List Program Improved 4](#_Toc96159539)

[4.1 Loading existing tasks into Memory 4](#_Toc96159540)

[4.2 Print current tasks and options menu 5](#_Toc96159541)

[4.3 Add or remove a task from the task list in memory 5](#_Toc96159542)

[4.4 Save or reload data to/from the file 6](#_Toc96159543)

[5 Summary 7](#_Toc96159544)

# Introduction

This write up will discuss how functions are used in Python, how to pass arguments to functions, the difference between arguments and parameters, and how organizing scripts by placing functions in large sections called classes will improve the previous assignment’s To Do List script.

# Functions

By definition, “A function is a block of code which only runs when it is called” (w3schools, <https://www.w3schools.com/python/python_functions.asp>, Feb,2022)(External Site). Functions are useful for allowing a script to perform a specific task the same way each time it is run. To run a function, it must be ‘called’. For a function to be called, it must be defined previously in the script. Listing 1 shows a simple function being defined and called with an argument.

def test\_function(name):  
 print("Hi, " + name)  
 return  
test\_function("Jerry Seinfeld")

*Listing 1 test\_function defined and called with an argument*

Text

Description automatically generated

*Figure 1 Output of script calling test\_function*

Figure 1 shows the output of the script. Functions are defined with the ‘def’ prefix before the function name. Arguments are any information that is passed into the function. Test\_function is passed the argument “Jerry Seinfeld”. The parameter ‘name’ inside the brackets is the variable that operates in the body of the function. Arguments are passed to functions and the parameters may take on the value of the argument and used inside the function. Functions may return something upon completion, however test\_function returns nothing. The To Do List program often takes in a list as an argument to a function, modifies the list, and then returns the modified list.

# Classes

## Organization

Organizing functions into meaningful sections of the code is very helpful. “A class is a way to take a grouping of functions and data and place them inside a container so you can access them with the . (dot) operator.” (Shaw, Zed A., Learn Python 3 The Hard way, Nov 2018). The To Do List has a large section of functions that perform processing tasks, and another section that performs input/output (IO) tasks, so it makes sense to create two classes called Processor and IO. Calling an IO function would look something like this IO.function(). See Figure 2.

Text

Description automatically generated

*Figure 2 Functions grouped inside classes*

## Separation of concerns

Separation of concerns is a technique that programmers use to organize their code in a meaningful, readable, and useful way. An example of this is creating a script with a section at the top for global variables, variable creation/initialization, and constants, then a section for processing which includes functions grouped inside classes, then a section for IO, then the main body of the program which prompts the user, calls functions, and operates until the code is complete. Organizing a script in this fashion is best practice, as it makes what is happening easier for other programmers to understand, makes faulty code easier to troubleshoot, and makes it easier to build the script up when beginning a project.

# To Do List Program Improved

The previous version of the To Do List program used a while loop to jump to different sections of code to perform certain tasks but did not utilize any functions. The new version delegates these repetitive lines of code to functions and groups functions into classes. The program begins by declaring and initializing variables, then defines all the functions within the two classes, Processor and IO, loads data from a file, then the main body of the program utilizes a While loop to prompt a user for an option involving adding or removing tasks, saving to the file or ending the program. The full code can be viewed here (<https://github.com/ryanlupinski/IntroToProg-Python-Mod06>)(external site).

## Loading existing tasks into Memory

The first step in the main body of the program calls function in the Processor class using Processor.read\_data\_from\_file(strFileName, lstTable) and passes two arguments, a string of the File Name, and a list table. See Listing 2.

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

# Processing ------------------------------------------------------------- #  
class Processor:  
 *""" Performs Processing tasks """* @staticmethod  
 def read\_data\_from\_file(file\_name, list\_of\_rows):  
 *""" Reads data from a file into a list of dictionary rows* ***:param*** *file\_name: (string) with name of file:* ***:param*** *list\_of\_rows: (list) you want filled with file data:* ***:return****: (list) of dictionary rows  
 """* list\_of\_rows.clear() # clear current data  
 try: # check if ToDoList.txt already exists  
 objFile = open(file\_name, "r") # open ToDoList.txt in read mode  
 for line in objFile: # for each line in the file  
 task, priority = line.split(",") # split elements in the file line by a comma, save to variables  
 row = {"Task": task.strip(), "Priority": priority.strip()} # create dicRow from previous variables  
 list\_of\_rows.append(row) # append each row to the lstTable  
 objFile.close() # close the file  
 except: # if ToDoList.txt doesn't exist create a new txt file and populate with a header  
 objFile = open(file\_name, 'w') # open the file  
 objFile.write("Task" + ',' + "Priority" + '\n') # write task,priority header to file  
 objFile.close() # close the file  
 print("ToDoList.txt not found. Creating a new ToDoList.txt file") # print to console  
 return list\_of\_rows, 'Success' # return the lstTable

*Listing 2 Processor.read\_data\_from\_file(strFileName, lstTable) section*

Arguments strFileName and lstTable are passed to the functions and assigned to the parameters file\_name and list\_of\_rows, respectively. A header section in triple quotes gives a description of each parameter. The same try-except method from the previous assignment is utilized to see if a file named ToDoList.txt already exists. See the comments in Listing 2 that describe function read\_data\_from\_file and what it does.

## Print current tasks and options menu

Now that the data from the file has been loaded into memory with the return list\_of\_rows line from the code in Listing 2, the tasks can be printed to the screen and along with a menu using the following code.

# Step 2 - Display a menu of choices to the user  
while True:  
 # Step 3 Show current data  
 IO.print\_current\_Tasks\_in\_list(lstTable) # Show current data in the list/table  
 IO.print\_menu\_Tasks() # Shows menu  
 strChoice = IO.input\_menu\_choice() # Get menu option

*Listing 3 While loop, print tasks, print menu, and prompt user for option code*

In the While loop, the task list table is passed to function print\_current\_Tasks\_in\_list which formats the tasks in a readable fashion. Next the print\_menu\_Tasks function displays a menu of options and returns nothing. The next line of code calls the function input\_menu\_choice and returns a string value choice which is saved to the variable strChoice. Now the if/elif/else statements will execute and perform the requested function.

## Add or remove a task from the task list in memory

If the user types ‘1’ they are prompted to enter a task and a priority. If they enter ‘2’, they are prompted to type in the name of a task to be removed from the list which is stored in memory as variables. See listing 4.

if strChoice.strip() == '1': # Add a new Task  
 strTask = str(input("Add task: ")) # prompt user to add a task  
 strPriority = str(input("Add priority[1(high) - 5(low): ")) # prompt user to define priority  
 Processor.add\_data\_to\_list(strTask, strPriority, lstTable) # pass task, priority, and task table args to add data function  
 IO.input\_press\_to\_continue(strStatus)  
 continue # to show the menu  
  
elif strChoice == '2': # Remove an existing Task  
 strTask = str(input("Remove a task: ")) # prompt user to add a task  
 Processor.remove\_data\_from\_list(strTask, lstTable) # pass task and task table as args to remove data function  
 IO.input\_press\_to\_continue(strStatus)  
 continue # to show the menu

*Listing 4 Add and remove option function call section*

Listing 4 shows arguments being passed to various functions depending on the option being chosen. Listing 5 shows the function definition in the Processor class section.

@staticmethod  
def add\_data\_to\_list(task, priority, list\_of\_rows):  
 dicRow = {'Task': task, 'Priority': priority} # create a new dicRow with new task  
 list\_of\_rows.append(dicRow) # append the table w/ new dicRow  
 return list\_of\_rows, 'Success' # return the updated list  
  
@staticmethod  
def remove\_data\_from\_list(task, list\_of\_rows):  
 for row in range(len(list\_of\_rows)): # look in each row of the lstTable  
 if list\_of\_rows[row]['Task'] == task: # check if task key equals the task to be removed  
 del list\_of\_rows[row] # delete the row  
 break # break the if loop  
 return list\_of\_rows, 'Success'

*Listing 5 Add and remove function definition*

Adding a task involves passing the task and priority to the function as arguments, creating a new dictionary row, and appending the list table of to do task. Finally, the updated task list is returned. Removing a task applies the same idea. The function is called, and the task and task list are passed as arguments. Each row in the list is looped through using the For function, and if the parameter variable task matches the ‘Task’ key in the dictionary row, the row is removed.

## Save or reload data to/from the file

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(strFileName, lstTable) # pass file name and task table to write to file function  
 IO.input\_press\_to\_continue(strStatus)  
 else:  
 IO.input\_press\_to\_continue("Save Cancelled!")  
 continue # to show the menu  
  
elif strChoice == '4': # Reload Data from File  
 print("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) # calls read file data function again  
 IO.input\_press\_to\_continue(strStatus)  
 else:  
 IO.input\_press\_to\_continue("File Reload Cancelled!")  
 continue # to show the menu

*Listing 5 Save data to file and load data from file*

Listing 5 shows what functions are called and the arguments that are passed. It is important to note that each of these options gives the user the ability to abort saving or reloading data. For instance, if the user forgot to add a task, they can skip saving and return to the menu. If the user forgot to save all the updated tasks in the task list in memory, they may not want to reload the tasks from the file, which would overwrite the current task list table in memory. Each function passes the file name and list table to the function as arguments. Option 4 has been covered in section 4.1 and will not be shown in this section.

@staticmethod  
def write\_data\_to\_file(file\_name, list\_of\_rows):  
 objFile = open(file\_name, 'w') # open connection to file in write mode  
 for row in list\_of\_rows: # for each row lstTable  
 strTask = row['Task'] # pull task value from lstTable  
 strPriority = row['Priority'] # pull priority value from lstTable  
 objFile.write(str(strTask) + "," + str(strPriority) + "\n") # write task/priority values to file  
 objFile.close() # close the file  
 return list\_of\_rows, 'Success'

*Listing 6 Save data to file function*

When the user is satisfied with the added or removed tasks in the task list in memory, they can save the updated task list to the file. A connection to the file is opened and for each row of the task list, a task and a priority are written to the objFile. The connection is closed and the list of rows is once again returned. Option 5 prints “Goodbye!”, breaks the While loop and the program ends.

# Summary

Functions are important in programming to keep code consistent and organized. When there are multiple functions that have different general purposes, they can be grouped into Classes. The improved To Do List program utilizes a Processor and IO Class to group functions that either process information or request input or print an output. Separating different sections of code to specific areas is part of the separation of concerns style of programming. Separating code into sections makes it easier to read, helps others who did not write the program to understand what is happening, and makes troubleshooting and organization simpler.

Text

Description automatically generatedText

Description automatically generated

*Figure 3 Assignment06\_RLupinski.py running in a terminal session*