

**Name:** Joseph Géigel

**Date:** May 24, 2023

**Course:** Foundations of Programming: Python

**Assignment:** 06

**GitHub URL:** <https://github.com/jgeigeluw/IntroToProg-Python/edit/main/docs/index.md>

# Improving the To Do List Script: Functions

## Introduction

The last two papers have introduced and discuss a script that generates a text file with a list of items defined by the user. We have made modifications to that script by adding Python elements to make it more efficient and better organized. Initially, we explored the concept of loops, specifically while loops, to allow the user to select an option that would cause the program to conduct an operation. Then we introduced dictionaries to associate tasks and priority levels in a to do list. This time we discuss the concept of functions, and how to implement them to group statements in Python and break up code into sections to make it easier to understand and maintain.

## Functions

The scripts that we have discussed so far have been relatively simple and short, and they only involve a few operations to obtain the desired output. However, in more complex scripts, it is useful to organize statements into sections and separate the parts of a script. Functions allow the programmer to break up code into manageable pieces that are easier to write, understand, and maintain. (Dawson, 2010) With functions the programmer can group one or more statements, and then instruct the program to perform an action by “calling” that function. (Root, n.d.)

### Defining and calling a function

When working with functions in Python, the first step involves defining the function. The action of defining a function tells the computer that the block of code that follows the definition statement is supposed to be used together as the function. (Dawson, 2010) Defining the function begins with a single line of code with the following syntax:

```
def function():
```

Where `def` is the keyword that establishes that a function will be defined, followed by the name of the function, a pair of parentheses, and a colon. After this statement, the indented lines that follow complete the function definition by specifying what are the actions to be executed when the function is called. The definition line and its block are the function, but no

action will be executed by the program until the function is called. Calling the function is done in the same way that built-in functions, such as `print()`, are called. The programmer enters a line that contains the name of the function, followed by its name and parentheses. These parentheses can contain parameters that are passed into the function.

### Parameters and arguments

When a function is called, parameters catch values sent to the function through arguments. (Dawson, 2010) The values passed into parameters are called arguments, and they are used in the function to perform actions. In figure 1, a function called `AgeCalc` calculates age in years by subtracting `birthYear` from `currYear`, and the arguments specifying the birth and current years are included within the parentheses when the function is called.

```
1  # Define function
2  def AgeCalc(currYear, birthYear):
3      yourAge = currYear - birthYear
4      print("Your age is: " + str(yourAge))
5
6
7  # Call function
8  AgeCalc(2023, 1992)
```

Figure 1. Example of a function defined and called.

Notice how the two parameters are included when the function is defined, and these two parameters are separated by commas. Functions can contain many parameters and there is no practical limit to the number of parameters in a function. (Root, n.d.) The mathematical operation conducted by the function is included in its definition, and it is followed by a print statement to display a string that reads “Your age is:”, followed by the result of subtracting the two values. When the function is called, actual arguments take the place of the parameters used in the function definition: current year is 2023, and birth year is 1992. The function then takes these argument values and performs the operation to calculate age, and the result is shown in figure 2.

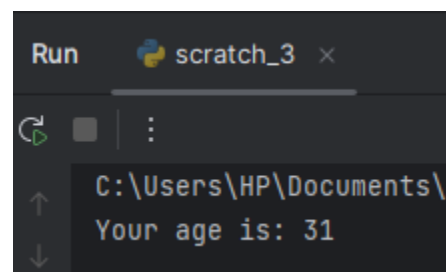
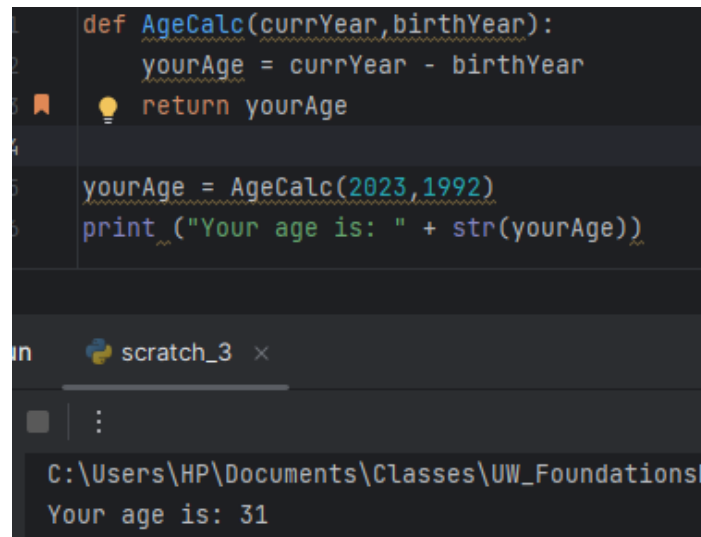
A screenshot of a Python IDE window titled 'scratch\_3'. The window shows a console output area with the text 'Your age is: 31'. The file path 'C:\Users\HP\Documents\O' is visible above the output. The window also has a 'Run' button and a Python logo icon.

Figure 2. Result from running the function defined and called in figure 1.

## Return values

The `return` keyword can be used in a function to capture a value and pass it back to the part of the program that called it, and the function ends. (Dawson, 2010) Using the `return` statement is a way of catching the result value of a function, and it allows the program to use that function as an expression that can be assigned to a variable. In the example below, a `return` statement was added to the `AgeCalc` function that we introduced before, and the function was used to define the variable `yourAge`. Then the variable was used to execute the function, evaluate 2023-1992, and print "Your age is: 31".



```
def AgeCalc(currYear, birthYear):  
    yourAge = currYear - birthYear  
    return yourAge  
  
yourAge = AgeCalc(2023, 1992)  
print("Your age is: " + str(yourAge))
```

The screenshot shows a code editor window titled 'scratch\_3'. The code defines a function `AgeCalc` that takes `currYear` and `birthYear` as arguments, calculates the age by subtracting `birthYear` from `currYear`, and returns the result. Below the function definition, the function is called with `AgeCalc(2023, 1992)` and the result is stored in `yourAge`. Finally, the result is printed using `print("Your age is: " + str(yourAge))`. The output of the script is visible at the bottom: `Your age is: 31`.

Figure 3. An example of a return statement being used to capture the result of a function.

## Classes

Classes allow programmers to group functions, variables, and constants. (Root, n.d.) This is helpful to build script that are more organized and easier to read, write and modify. The use of classes is convenient, and it helps the programmer to adhere to the principle of separation of concerns in programming.

## Separation of Concerns

The concept of separation of concerns refers to a principle in computer programming where code is separated in sections based on the purpose of the code. This design approach facilitates the organization of work in a more effective way. A complex script can be separated into distinct sections that accomplish different tasks, such as: storing the data, processing the data, or displaying data to the human user. Functions and classes facilitate the separation of concerns when writing code, because functions can group and organize multiple statements to perform a task, and these functions can also be grouped into classes to separate the code.

## The Enhanced To Do List Script

In the previous paper, we used dictionaries to associate tasks and priority levels in a to do list. This time we discuss functions and classes, and how to implement them to group statements in Python and break up code into sections to make it easier to understand and maintain. The development of this script followed the principles of separation of concerns to divide the code into four sections:

- Data
- Processing
- Presentation
- Main body

The processing and presentation sections of the code include functions that are grouped in two classes named: *Processor* and *IO*.

### Data

As it was done in the previous version of the To Do List script, the data section declares variables that will be necessary to perform several tasks. Figure 4 presents the five variables included in the new version of the To Do List script. The first one is `file_name_str`, a variable that contains the name of the text file that will be read and edited by the script. The second variable is `file_obj`, and it is initiated as a placeholder for the file object. The third variable is an empty dictionary that will contain tasks and priorities as captured from the file or from user input. The fourth variable is `table_lst`, a list that acts as a table of tasks and priorities. Finally, the `choice_str` variable is initiated to capture the user option selection that will guide the execution of the script.

```
13 #----- Data -----
14 # Declare variables and constants
15 file_name_str = "ToDoList.txt" # The name of the data file
16 file_obj = None # An object that represents a file
17 row_dic = {} # A row of data separated into elements of a d
18 table_lst = [] # A list that acts as a 'table' of rows
19 choice_str = "" # Captures the user option selection
20
```

Figure 4. Data section that declares variables and constants in the script.

### Processing: The Processor Class

Once variables and constants are declared in the *Data* section of the script, the *Processing* section is introduced. This section contains one class named *Processor* with four functions that are used to perform processing tasks in the To Do List script. The first of these functions is `read_data_from_file`, and it reads data from the text file and iterates through each line using a for loop. This for loop is used to create a list of tasks and priorities for each line in the

file object, and the items in these lists are used to define key-value pairs in the dictionary. Those dictionary key-value pairs are then appended as tasks and priorities in a list.

The function takes two arguments: a file name and a list of rows. The values for the two parameters are specified in the main body of the To Do List script with variables:

`file_name_str` and `table_lst`, and they are the to do list text file, and the table returned from reading the lines in the file object. As shown in figure 5, this function is executed as soon as the program starts, and it essentially loads any data present in the existing to do list.

```
23 class Processor:
24     """ This class contains the functions that perform processing tasks """
25     1 usage
26     @staticmethod
27     def read_data_from_file(file_name, list_of_rows):
28         """ Reads data from a file into a list of dictionary rows
29         :param file_name: (string) with name of file:
30         :param list_of_rows: (list) you want filled with file data:
31         :return: (list) of dictionary rows
32         """
33         list_of_rows.clear() # clear current data
34         file = open(file_name, "r")
35         for line in file:
36             task, priority = line.split(",")
37             row = {"Task": task.strip(), "Priority": priority.strip()}
38             list_of_rows.append(row)
39         file.close()
40         return list_of_rows
```

Figure 5. The `read_data_from_file` function reads data from a text file and uses the rows to create a table of tasks and priorities.

```
#----- Main Body of Script -----#
# Step 1 - When the program starts, Load data from ToDoList.txt.
Processor.read_data_from_file( file_name=file_name_str, list_of_rows=table_lst) # read file data
```

Figure 6. Execution of the `read_data_from_file` function as the first step in the main body of the script.

The next three functions in the processing section of our script complete three different tasks: adding new tasks and priorities, removing tasks from the list, or writing the data to the file object. The execution of these three functions is dependent on the choice made by the user when the program runs.

The first function is `add_data_to_list` and it accomplishes the simple task of populating key-value (task-priority) pairs in a dictionary, and then using the data in that dictionary to append new data to the list of rows. The function takes three arguments: a string representing a task, a string for a priority value, and the list of rows.

```
def add_data_to_list(task, priority, list_of_rows):
    """ Adds data to a list of dictionary rows
    :param task: (string) with name of task:
    :param priority: (string) with name of priority:
    :param list_of_rows: (list) you want to add more data to:
    :return: (list) of dictionary rows
    """
    row = {"Task": str(task).strip(), "Priority": str(priority).strip()}
    list_of_rows.append(row)
    return list_of_rows
```

Figure 7. This function is used to add new tasks and priorities to the to do list.

After this function, the next one in the processor class is tasked with removing tasks and priorities from the list. The `remove_data_from_list` function takes two arguments: a string specifying the task to be removed, and the list of rows. In order to remove a task from the list, the function uses a for loop to find the task to be removed, and then the `remove` list method is used to take the row off the to do list.

```
def remove_data_from_list(task, list_of_rows):
    """ Removes data from a list of dictionary rows
    :param task: (string) with name of task:
    :param list_of_rows: (list) you want filled with file data:
    :return: (list) of dictionary rows
    """
    for row in list_of_rows:
        if row["Task"].lower() == task.lower():
            list_of_rows.remove(row)
    return list_of_rows
```

Figure 8. This function removes a specified task from the to do list.

The last function in the processing function of the script allows the user to save the data to the text file object. The `write_data_to_file` function opens the file in write mode and uses the `write` method to write new lines in the text file. After writing down all the items, the function closes the file object.

```
def write_data_to_file(file_name, list_of_rows):
    """ Writes data from a list of dictionary rows to a File
    :param file_name: (string) with name of file:
    :param list_of_rows: (list) you want filled with file data:
    :return: (list) of dictionary rows
    """
    file = open(file_name, "w")
    for row in list_of_rows:
        file.write(row["Task"] + "," + row["Priority"] + "\n")
    file.close()
    return list_of_rows
```

Figure 9. The last of function of the processing section. This function writes data to the text file.

### Presentation (Input/Output): The IO Class

After the Processor class defines all the functions that are necessary to process the input data, the IO (input-output) class groups five functions that perform input and output tasks in the script. This class will present data to the user, and it will also capture input data that is required to execute the functions in the Processor class.

The first function in this class is the `output_menu_tasks` function (figure 10), and its only purpose is to print a menu of options for the user. The menu of options is the very similar to the menu that was presented in the previous version of the to do list script. It displays numbers from 1 to 4 that identify different options that the user can choose to instruct the program to perform different actions.

```
class IO:
    """ This class contains functions that perform Input and Output tasks """
    1 usage
    @staticmethod
    def output_menu_tasks():
        """ Display a menu of choices to the user
        :return: nothing
        """
        print('''
        Menu of Options
        1) Add a new Task
        2) Remove an existing Task
        3) Save Data to File
        4) Exit Program
        ''')
        print() # Add an extra line for looks
```

Figure 10. The output menu tasks function displays a menu of choices to the user.

With the possible options presented to the user, the next function in the IO class is tasked with capturing an input string that represents the choice made by the user. Notice that the only task accomplished by this function is to capture that information, it is not a processing function, so it

only captures the data that is required by a `while` loop in the main body of the script to determine which processing functions to execute.

```
def input_menu_choice():  
    """ Gets the menu choice from a user  
    :return: string  
    """  
  
    choice = str(input("Which option would you like to perform? [1 to 4] - ")).strip()  
    print() # Add an extra line for looks  
    return choice
```

Figure 11. The input menu choice function captures a string value that represents the choice made by the user.

The next presentation function is the `output_current_tasks_in_list` function (figure 12). This one takes one argument, the list of rows, and its only purpose is to print the current tasks and priorities in the to do list. To do this, the function uses a for loop that iterates over the list of rows and prints each row.

```
def output_current_tasks_in_list(list_of_rows):  
    """ Shows the current Tasks in the list of dictionaries rows  
    :param list_of_rows: (list) of rows you want to display  
    :return: nothing  
    """  
  
    print("***** The current tasks ToDo are: *****")  
    for row in list_of_rows:  
        print(row["Task"] + " (" + row["Priority"] + ")")  
    print("*****")  
    print() # Add an extra line for looks
```

Figure 12. The output current tasks in list function prints the tasks in the to do list.

Finally, we have the last two input functions, the `input_new_task_and_priority` and `input_task_to_remove` functions. These two functions (shown in figure 13) accomplish very similar tasks, they capture the string data that is necessary to identify tasks and priorities to be added or removed by the processing functions that were defined to perform those actions. Again, the purpose of these presentation functions is limited to capturing data necessary to perform the actions of processing functions.



```

def input_new_task_and_priority():
    """ Gets task and priority values to be added to the list
    :return: (string, string) with task and priority
    """
    task = str(input("What is the task? - ")).strip()
    priority = str(input("What is the priority? [high|low] - ")).strip()
    return task, priority

1 usage
@staticmethod
def input_task_to_remove():
    """ Gets the task name to be removed from the list
    :return: (string) with task
    """
    task = str(input("What is the name of task you wish to remove? - ")).strip()
    print() # Add an extra line for looks
    return task

```

Figure 13. These two functions are used to capture tasks to be added or removed.

### Main Body of the Script

With all processing and presentation functions defined, the development of the body of the script is straightforward. All the actions executed by the script can be grouped in four main steps:

- Load the data
- Display the menu
- Show current data
- Process user choices

As seen in step 1 in figure 14, the first action taken in the main body of the script involves calling the `read_data_from_file` function from the `Processor` class. Functions are called by the program by using a syntax where the class name is followed by a dot, and then the function is called by its name, and finally the parameters within the parentheses are assigned their respective arguments. In this case, the two arguments for the `file_name` and `list_of_rows` parameters are specified by the `file_name_str` and the `table_lst` variables respectively. The execution of this function in the main body of the script loads any tasks and priorities present in the existing file object.

```

#----- Main Body of Script ----- #

# Step 1 - When the program starts, Load data from ToDoList.txt.
Processor.read_data_from_file( file_name=file_name_str, list_of_rows=table_lst) # read file data

# Step 2 - Display a menu of choices to the user
while (True):
    # Step 3 Show current data
    IO.output_current_tasks_in_list(list_of_rows=table_lst) # Show current data in the list/table
    IO.output_menu_tasks() # Shows menu
    choice_str = IO.input_menu_choice() # Get menu option

    # Step 4 - Process user's menu choice
    if choice_str.strip() == '1': # Add a new Task
        task, priority = IO.input_new_task_and_priority()
        table_lst = Processor.add_data_to_list(task=task, priority=priority, list_of_rows=table_lst)
        continue # to show the menu

    elif choice_str == '2': # Remove an existing Task
        task = IO.input_task_to_remove()
        table_lst = Processor.remove_data_from_list(task=task, list_of_rows=table_lst)
        continue # to show the menu

    elif choice_str == '3': # Save Data to File
        table_lst = Processor.write_data_to_file(file_name=file_name_str, list_of_rows=table_lst)
        print("Data Saved!")
        continue # to show the menu

    elif choice_str == '4': # Exit Program
        print("Goodbye!")
        break # by exiting loop

```

Figure 14. Whole main body of the script after classes and functions are defined.

With the existing data loaded to memory, the program begins a while loop that will use if and elif statements to perform actions based on the choices made by the user. The first step in the selection of these choices is the presentation of current data in the list, as well as the menu of possible choices and the prompt to get a choice from the user. These actions are completed by the three input/output functions called in step 3. Notice how the `IO.input_menu_choice` function is used to define the `choice_str` variable. This variable is a key element of the script because it is used to evaluate if conditions are met to execute processing functions in step 4.

Step 4 in figure 14 shows all the conditional statements that guide processing in the script. If the option chosen by the user matches the choice string in the conditional statement, then the appropriate function will be executed. This is a summary of the actions performed by the script for each option:

1. If the user chooses option 1, the `input_new_task_and_priority` function will run to prompt the user to enter a new task and priority value, and then the `add_data_to_list` function appends the entered values to the list.
2. If option 2 is chosen, the `input_task_to_remove` runs to prompt the user to enter a task to remove, and the processor `remove_data_from_list` function removes the selected task from the list.
3. Option 3 runs the `write_data_to_file` function, which saves all the data to the text file object.
4. Lastly, option 4 exits the program by breaking out of the while loop.

Notice how after the execution of each option, except for option 4, a `continue` statement makes the program display the menu of options again. This allows the user to make additional changes to the to do list before exiting the program.

#### Output example

Most of the changes applied to the To Do List script in this new version are related to background processes that do not have significant impacts on the output data. Output from this improved version of the script is similar to what was obtained from running the previous version of the script. Figure 15 shows an example of running the script in PyCharm, where we see the current list of tasks, followed by the menu of options. In this example, the user selected option 1, and was prompted to enter a task and a priority level. Option 1 added these values to list, and the updated list was displayed.

```

***** The current tasks ToDo are: *****
read chapter (high)
play Civilization (low)
take a nap (low)
eat hot chip (high)
go for a walk (low)
Cook dinner (high)
Run (high)
*****

Menu of Options
1) Add a new Task
2) Remove an existing Task
3) Save Data to File
4) Exit Program

Which option would you like to perform? [1 to 4] - 1

What is the task? - complete assignment
What is the priority? [high|low] - high
***** The current tasks ToDo are: *****
read chapter (high)
play Civilization (low)
take a nap (low)
eat hot chip (high)
go for a walk (low)
Cook dinner (high)
Run (high)
complete assignment (high)
*****

```

Figure 15. Example from running the script in PyCharm and selecting option 1.

## Summary

In this paper, several adjustments were made to the To Do List script. The main modification to this script consisted in the introduction of classes and functions to improve the organization of the code. This modification follows the design philosophy of separation of concerns, where code is divided in sections according to the actions executed by each section. Following these principles, the To Do List script was subdivided into four sections: data, processing, presentation, and main body. The statements in the processing and presentation sections were grouped into classes that contained functions that displayed data, captured input data from the user, and processed data. This reorganization of the code resulted in an well-organized script that is easier to understand and modify.

## References

Dawson, M. (2010). *Python Programming for the Absolute Beginner* (Third Edition ed.). Boston, MA, United States of America: Course Technology PTR.

Root, R. (n.d.). Programming with Python: Module 06.