Mark Brugger
November 23, 2022
IT FDN 110 A – Foundations of Programming: Python
Assignment06

# A Classy To Do List

## Introduction

This week is a continuation of the previous to-do list script. Once again, we were given a starter python file. The objective was to add classes and functions to the script in order to better follow the principle of Separation of Concerns.

## Learning concepts this week

Custom functions were a big portion of the new topics this week. With a function, you can group one or more statements and Python will run them when you call the function:

```
Function_name()
```

It is possible to pass arguments in your functions too, just like when you run one of Python's built-in statements.

```
Function_name(input1, input2)
```

Functions are defined at the top of the script, prior to the main body. While they reside up top, they will not be run until they are called in the main body of the script. In fact, it is possible to define multiple functions that never get run, though that wouldn't be very useful.

When a function is done running, it can return a value or multiple values back to the main part of the script. As an example, if there was a function whose job it was to perform some mathematical calculations of two numbers that were input, the function would then return the answer.

When initializing variables, an empty string "" is often used for string variables. A [] or {} is the equivalent for lists and dictionaries. Instead, we learned we could start with ***None*** which represents the absence of any value.

Speaking of variables, we were introduced to the concept of global and local variables. Thus far we have been working with global variables. It is also possible to define variables inside a

function.  They are considered local because they are only recognized there.  If I defined the variable input_file inside a local variable and then tried to call it from the main body, Python would return an error.  Additionally, it is possible to use the same name for a global variable and a local variable and have them exist independently.

Finally, we worked with classes, which organize groups of functions, variables, and constants. In our script this week, we have a Processor and an IO (input/output) class to clearly segregate our functions into the category of performing processing tasks and a second category to receive input from the user and provide a response.

## Process to make program

We begin this week's script with the usual header:

```
# ------------------------------------------------------------------------ #
# Title: Assignment 06
# Description: Working with functions in a class,
#              When the program starts, load each "row" of data
#              in "ToDoToDoList.txt" into a python Dictionary.
#              Add the each dictionary "row" to a python list "table"
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# Mark Brugger,11.21.2222,Modified code to complete assignment 06
# ------------------------------------------------------------------------ #
```

*Figure 1 - Script Header*

All the global variables (shown in Figure 2) were defined for us in the starter code.  For the first time, the external text file was defined in a string.  As the file name is used in multiple locations, this will make updating the code easier in the future, as well as preventing potential errors from a typo somewhere.

```
# Data -------------------------------------------------------------------- #
# Declare variables and constants
file_name_str = "ToDoFile.txt"  # The name of the data file
file_obj = None  # An object that represents a file
row_dic = {}  # A row of data separated into elements of a dictionary {Task,Priority}
table_lst = []  # A list that acts as a 'table' of rows
choice_str = ""  # Captures the user option selection
```

*Figure 2 - List of global variables*

The first set of functions is contained within the Processor class. Inside, the first function is read_data_from_file, which as one would expect, imports task items from an external file if it exists. The code was mostly there already; however, it would cause a Python run-time error if ToDoFile.txt did not already exist. To fix this, I added a try/except loop (see Figure 3). The existing code went into the try section, and the except printed a message to the user and allowed the script to keep running. If the text file does exist, the function imports the tasks and priorities into list for future use by the script.

```
# 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:

            file = open(file_name, "r")
            for line in file:
                task, priority = line.split(",")
                row = {"Task": task.strip(), "Priority": priority.strip()}
                list_of_rows.append(row)
            file.close()
            return list_of_rows

        except:

            print("\nFile not found. One will be created when you save.\n")
            return list_of_rows
```

*Figure 3 - The read_from_file function*

The next few functions in the Processor class lack some context until you look at the functions in the IO class. Explanations will be added to avoid reader confusion.

The add_data_to_list function, shown in Figure 4, takes user input to add a new task and its priority level to the list of dictionary rows. A function in the IO class solicits the task information and then passes that data to this function. My contribution to the function was adding the append command. At the end of the function, the new list_of_rows list is passed back to the main part of the script.

```
@staticmethod
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 filled with file data:
    :return: (list) of dictionary rows
    """
    row = {"Task": str(task).strip(), "Priority": str(priority).strip()}
    # TODO: Add Code Here!
    list_of_rows.append(row)
    return list_of_rows
```

*Figure 4 - The add_data_to_list function*

The remove_data_from_list function (see Figure 5) takes user input to delete a task from the list. A function in the IO class solicits the task name and then passes it to this function. Last week there was a bug in my code where it would display “Task was not found” as it was stepping through each line of the list of dictionary rows looking for a match of the task name. Ideally there would only be one message presented to the user, either confirmation that their task had been removed or a statement that the task was not found. To remedy that, I created a local Boolean variable called false_flag. I initialized it to be False initially. If the task was in the list, then false_flag was set to True. At the end of the for statement, an if statement was added to check to see if false_flag was still False. If so, then the user would be notified that their task was not found in the list. At the end, the function returns the list_of_rows, in its new state, back to the main section.

```
@staticmethod
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
    """
    # TODO: Add Code Here!

    found_flag = False
    for row in list_of_rows:
        if row["Task"].lower() == task.lower():
            list_of_rows.remove(row)
            print("\nYour task has been removed")
            found_flag = True
    if found_flag == False:
        print("\nTask was not found")

    return list_of_rows
```

*Figure 5 - The remove_data_from_list function*

The write_data_to_file function (see Figure 6) takes the list of dictionaries and writes it to an external text file. If ToDoList.txt already existed when the script was run, its entire contents were read into memory. Therefore, it is acceptable (and preferred) to use the "write" command when opening the file, rather than "append.

```python
@staticmethod
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
    """
    # TODO: Add Code Here!

    file = open(file_name, "w")
    for row in list_of_rows:
        file.write(str(row["Task"]) + "," + str(row["Priority"] + "\n"))
    file.close()

    return list_of_rows
```

*Figure 6 - The write_data_to_file function*

The next class, IO, performs input and output tasks for the user. The first function in the class, ouput_menu_tasks, displays a menu of choices (see Figure 7). It is purely a function to print information on the screen.

```python
class IO:
    """ Performs Input and Output tasks """

    @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 7 - The output_menu_tasks function*

The next function, input_menu_choice (see Figure 8) asks the user which selection they would like to make. That choice is passed back to the main part of the script.

```
@staticmethod
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 8 - The input_menu_choice function*

The output_current_tasks_in_list function, as shown in Figure 9 below, takes the list_of_rows list of tasks and priorities and prints it on the screen for the user.

```
@staticmethod
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 9 - The output_current_tasks_in_list function*

Next, we have input_new_task_and_priority. This is the function that asks the user to input the name and then the priority level of the task they want to add to their to do list. The function will return the task and priority back to the main part of the script and the script will then pass those to the function add_data_to_list which was described above.

```
@staticmethod
def input_new_task_and_priority():
    """  Gets task and priority values to be added to the list

    :return: (string, string) with task and priority
    """
    pass  # TODO: Add Code Here!

    task = input("Enter the name of the task: ")
    priority = input("Enter the priority level of the task, (Low/Medium/High): ")

    return task, priority
```

*Figure 10 - The function input_new_task_and_priority*

Finally, the input_task_to_remove function (see Figure 11) asks the user for the name of the task they want to remove from their to do list. The function takes the user's input and passes it to the main part of the script. The main part of the script sends this information to the function remove_data_from_list. Remove_data_from_list was covered previously in the document.

```
@staticmethod
def input_task_to_remove():
    """  Gets the task name to be removed from the list

    :return: (string) with task
    """
    pass  # TODO: Add Code Here!

    removal = input("Enter the name of the Task to Remove: ")

    return removal
```

*Figure 11 - The input_task_to_remove function*

By comparison, the main body of the script is relatively short. It starts with calling (as shown in Figure 12 below) the read_data_from_file function from the Processor class.

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

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

*Figure 12 - Importing previously saved to-do list items*

The second and final portion of the main body of the script is a while loop, shown in Figure 13, which directs the script to call different functions based on the choice the user made from the menu. The loop will continue until the user selects "4" at which point the code prints "Goodbye!" and terminates the script.

```
# 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 13 - While loop for the menu of choices*

When running the script in PyCharm IDE, the output is shown in Figures 14-16.

```
/Users/markbrugger/Documents/_PythonClass/Assignment06/venv/bin/python /Users/markbrugger/Documents/_PythonClass/Assignment06/Assigment06.py
******* The current tasks ToDo are: *******
Homework (High)
Wash Clothes (Low)
Put Away Clothes (Medium)
*****************************************


        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] - 2

Enter the name of the Task to Remove: Cook Dinner

Task was not found
```

*Figure 14 - Initial output from PyCharm*

```
*****************************************


        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

Enter the name of the task: Sleep
Enter the priority level of the task, (Low/Medium/High): High
******* The current tasks ToDo are: *******
Homework (High)
Wash Clothes (Low)
Put Away Clothes (Medium)
Sleep (High)
*****************************************
```

*Figure 15 - Output from Adding a task*

```
*****************************************


        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] - 3

Data Saved!
******* The current tasks ToDo are: *******
Homework (High)
Wash Clothes (Low)
Put Away Clothes (Medium)
Sleep (High)
*****************************************
```

*Figure 16 - Output from Saving the data to the external text file*

As expected, the script behaves the same (see Figure 17) when run from the MacOS Terminal window.

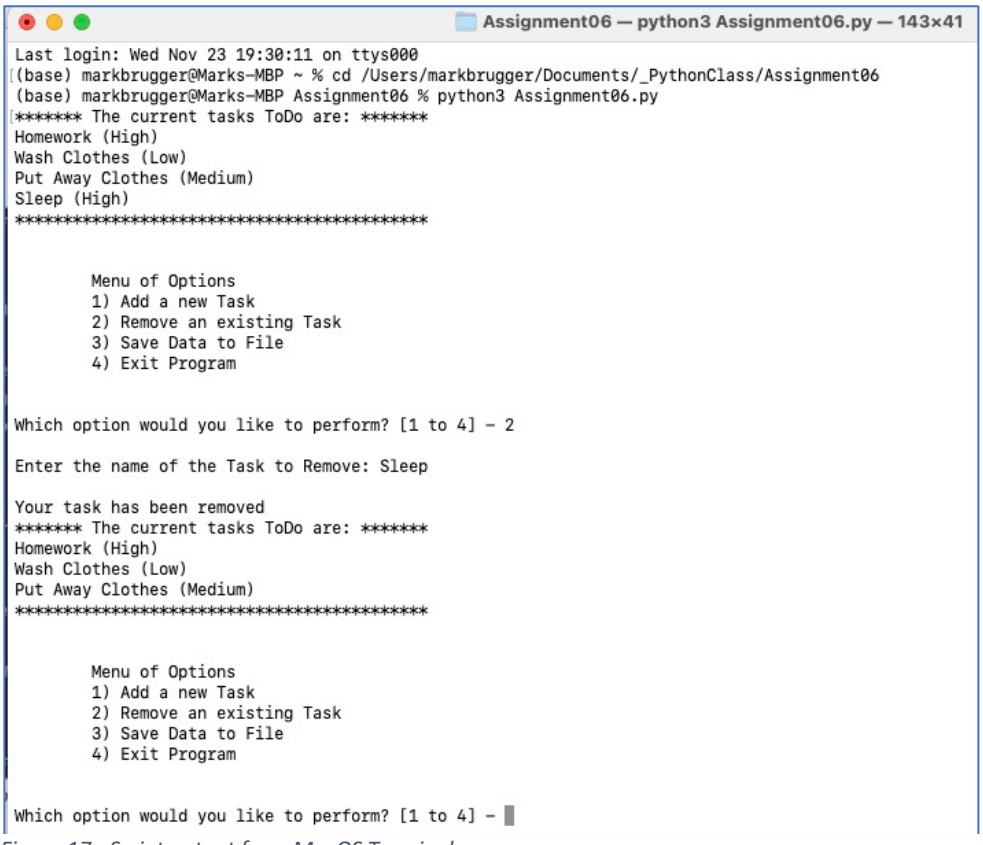


*Figure 17 - Script output from MacOS Terminal*

As a check, there is a ToDoFile.txt in the script folder and it contains the tasks that I saved.

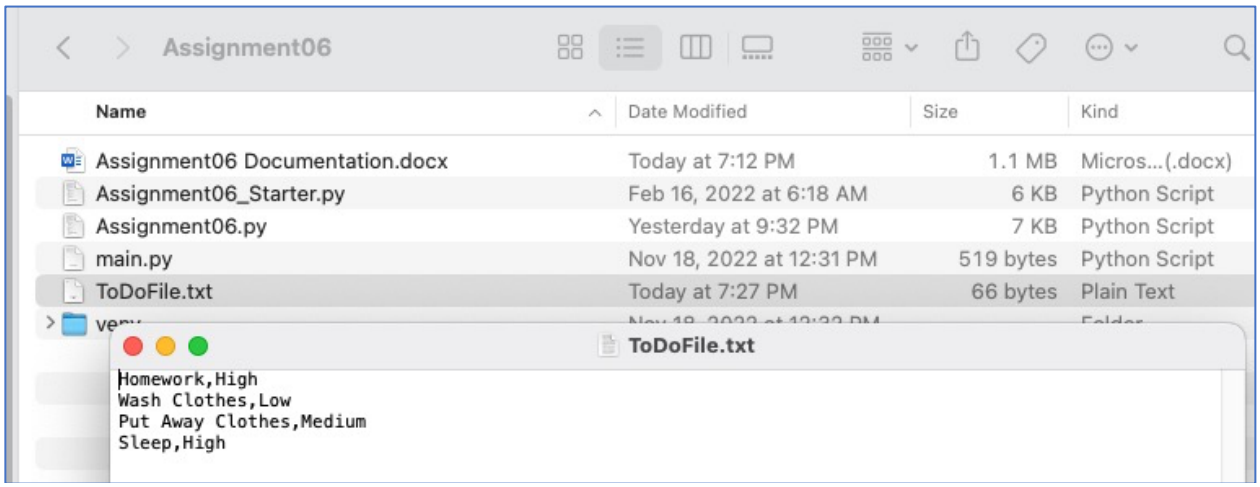


*Figure 18 - ToDoList.txt after saving the task data in the script*

## Summary

This week we started using functions extensively to help better organize a script. The To Do List script is now much more modular, and easier to modify in the future if desired.