Name: Kimberly Harms
Date: 8/9/2023
Course: IT FDN 110 A Su 23: Foundations of Programming: Python
Assignment 06
https://github.com/kimtara/IntroToProg-Python-Mod06

Functions

Introduction

For this assignment, the goal was to revisit the familiar To Do List program and write code that accomplished the same task using custom functions and classes. Functions are collections of statements that can be defined in a script and then called to execute the set of statements whenever needed. Classes are collections of functions, often a set of functions designed to perform similar types of tasks, such as mathematical functions or those handling the input and output. Using different classes of related functions helps with separation of concerns, a common practice for organizing code.

Defining Functions in the IO Class

I used the provided starter code for assignment06 which outlined two classes of functions, a processor class and an input-output or IO class. The IO class is a set of functions that contain all the code needed to get input from the user and provide instructions or information back to the user. Figure 1 shows the code that defines the function for getting new task and priority input from the user. When this function is called, it will do three things: get user task input and assign it to the task variable, get user priority input and assign it to the priority variable, and return the task and priority variables to use in subsequent code. Similarly, the function input_task_to_remove (figure 2) will do two things when called: ask the user for input to indicate which task to remove and assign that data to the task and return the task value to use in the next bit of code. This class also includes functions to display the menu of program options to the user, get the user's choice, and display the list of tasks and priorities to the user. Notice that each function can use the same variable names when they are local to that function. The return values (if any) are passed to the next statements in the main body of the code, as discussed below.

```
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()  # get task from user
    priority = str(input("What is the priority? [high|low] - ")).strip()  # get priority from
user
    return task, priority  # returns task and priority values
```

Figure 1. Function for getting new task and priority input from user.

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

    :return: (string) with task
    """
    # Added Code Here
    task = str(input("Which TASK would you like to remove? - ")).strip()  # get task from user
    return task  # pass out that task
```

Figure 2. Function that asks user which task to remove from ToDoList.

Defining Functions in the Processor Class

The other set of functions in the ToDoList program are in the processor class. These functions process the data as needed to complete the various pieces of the program. When the user provides a new task and priority, those data are put in a dictionary, which is in turn added to the list of dictionary rows (or table) with the add_data_to_list function (figure 3). Unlike the functions in figures 1 and 2, this function has three parameters required to perform its task – a task, priority, and list_of_rows. Other parts of the program provide the arguments passed into this function to meet the parameters required, which are processed in a list of dictionary rows. The function returns the final list_of_rows, and the program continues. Other functions in the processor class read data from a text file, deletes items from the To Do List (figure 4), and writes data to a text file (figure 5). This last function does not change the data, the list of rows passed in the argument is the same as that returned, but copies it to a file for storage.

```
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
    """

    # Added Code Here
    row = {"Task": str(task).strip(), "Priority": str(priority).strip()}  # create dictionary
    list_of_rows.append(row)  # add dictionary row to list
    return list_of_rows  # returns list with added items
```

Figure 3. Processing task, priority input into table list with the add_data_to_list function. This function has three parameters and returns a 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
    """
    # Added Code Here
    for row in list_of_rows:  # step through table of tasks
        old_task, old_priority = dict(row).values()  # pull out elements in dictionary rows
        if task == old_task:  # match user-entered task with existing tasks in dictionary rows
            list_of_rows.remove(row)  # remove row of matching task
    return list_of_rows  # return list of remaining items
```

Figure 4. Processing removal of task with remove_data_from_list function. This function has two parameters and returns a list.

```
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
    """
    # Added Code Here
    file_obj = open(file_name, 'w')
    for row in list_of_rows:
        file_obj.write(row["Task"] + "," + row["Priority"] + "\n")
    file_obj.close()
    return list_of_rows
```

Figure 5. Saving data to text file with write_data_to_file function.

Using Doc Strings

For each function defined, it is useful to include a doc string, or documentation for the person using the code. The doc string follows the name of the function, preceding the statements, and describes the purpose of the function, the parameters needed to execute it (if any), and what the function is expected to return (if anything) in human language. This is useful for the person using the code but is also useful when ordering the functions in the main body of the code. At each step I ask, 'what does this function need?' and 'what does this function provide?'. The element(s) provided should match what is needed by the next step.

The Main Body of the Script

The main body of the script dictates how the information flows from one piece of the program to the next. This piece of the script was provided in the starter code, so I did not do much here. However, I noted that when starting the program there is no ToDoList text file, which produces an error. To handle this error, I added in a try and except structure to step 1 of the main body. This lets the program load data when there is an existing text file but skips this step if no such file exists.

```
try:
# 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
except:
    print('Congratulations, your To Do List is empty! Would you like to add a task?')
```

Figure 6. Error handling with the try/except structure.

Testing the Program and Last Steps

Although I iteratively tested and debugged each section of the script as I went, the remaining piece of writing a program is to test it. I successfully added, deleted, and saved task and priority data to a text file (figure 7). I could repeat each of these options as needed until finally choosing to exit the program. I tested the script in both the PyCharm IDE (figure 8), and from the Command Prompt (figure 9). Finally, I uploaded my code to my GitHub repository (https://github.com/kimtara/IntroToProg-Python-Mod06).

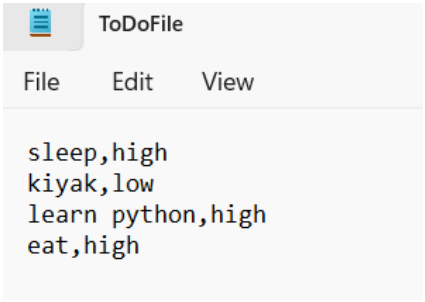

Figure 7. Example of ToDoList data saved to text file. Happily, all tasks here were achieved during this module.

```
What is the task? - learn python
What is the priority? [high|low] - high
******* The current tasks ToDo are: *******
sleep (high)
kiyak (low)
learn python (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] - 3

Data Saved!
******* The current tasks ToDo are: *******
sleep (high)
kiyak (low)
learn python (high)
*****************************************


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

Figure 8. Testing code in PyCharm IDLE. Successfully added, removed, and saved data. (Not all test output shown here.)

```
Command Prompt - python.e
What is the task? - snack
What is the priority? [high|low] - low
******* The current tasks ToDo are: *******
sleep (high)
kiyak (low)
learn python (high)
eat (high)
snack (low)
*****************************************


       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

Which TASK would you like to remove? - snack
******* The current tasks ToDo are: *******
sleep (high)
kiyak (low)
learn python (high)
eat (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] -
```

Figure 9. Testing code in Command Prompt. Successfully added, removed, and saved data. (Not all test output shown here.)

Summary

In this module I learned how to better achieve Separation of Concerns by creating and using classes of functions and custom functions. I can define functions, which are a collection of statements, to do a specific operation whenever called. When defining a function, I can require any number of parameters and what, if anything, is returned by the function. Each step of the program should provide the needed arguments for the function to execute. Functions can be divided into classes, or collections of functions, that have similar types of roles in the script. Then the main body of the script directs the flow from function to function as the program runs.

References

Randall Root, https://youtube.com/playlist?list=PLfycUyp06LG9fZlIlqBrxLcNV4CR50HEX, module 6.

Michael Dawson, Python Programming for the Absolute Beginner, 3rd Edition (2010).

Trevor Payne, Let's Learn Python Basics – Functions, https://www.youtube.com/watch?v=qO4ZN5uZSVg.