Patrick Regan
Aug 12th, 2023
ITFDN 110 A
Assignment 06
https://github.com/egg2020/IntroToProg-Python

Functions and Classes

Intro

This week we learned a bit about functions and classes. Functions are basically a way to create mini programs within a program, to be used over and over when needed. The variables used in a function are kept seperate from variables of the same name in the rest of the program, unless coded to be updated after the function ends. Classes are a way to organize functions, which helps conform with the rules of "Seperation of Concern".

Program

This code was mainly written for us, and we had to add code in to make this program work. For this document, I will describe how this code works, and where I ran into problems.

As always, at the top of the script is a header that tells reader about the code, as well as any modifications. Next is the Data section of our code, where global variables will be stored. These global variables will be used through out the code, and can be "seen" even within functions.

This code has four main parts of the script body. The Data section, which was just described above, then the Processing part of the code. This is where tasks are processed behind the scenes of what the user sees on the screen. Here I ran into an issue of trying to keep in line with the Seperation of Concern rules. Even though this is the Processing part of the code, I still felt it was better to put in a "print" statement (line 76) to let the user know the item had been removed. It may have been better to put this print statement in the "Processor.remove_data_from_list" function, but I would have needed to write complicated code to do so. After thinking about this, I am curious what the best way to deal with this is. Anyways, I did put this print function in the Processing part of the code. Moving on, the next part is the Presentation (Input/Output) section. This is where inputs and outputs are processed. In other words, this is where user prompting takes place, as well as user feedback. Last is the Main part of the script. This is where the code begins, where it calls on different Class/Functions to do work. When a function is called, values are returned to the main body, to be used with the next function called. Doing things in this way makes it easier to organize the coding, as well as keeping the coding as small as possible.

Without going through how the entire program works, I will describe the steps I took to

complete this assignment.

First thing I did was figure out how the code was inteded to work. I realized that the Class/Functions were only there to be used when called. So I found the main body of the script, where each function is called. I first started with line 173:

```
172  # Step 1 - When the program starts, Load data from ToDoFile.txt.
173  Processor.read_data_from_file(file_name=file_name_str, list_of_rows=table_
```

figure 1. first function call in program

This line calls on the read_data_from_file function. Arguments are passed along with this call, and they are set to equal existing variables. These arguments will be used in the function.

I went through this step in order for the main body of the program. Each time a function was called, I navigated to that function to see how it worked. The first function that I needed to fill out for the assignment was the following:

```
     1 usage
147  @staticmethod
148  def input_new_task_and_priority():
149      """  Gets task and priority values to be added to the list
150
151      :return: (string, string) with task and priority
152      """
153      pass  # TODO: Add Code Here!
154      task = str(input("what is the task name?")).strip() # ask for task name, s
155      priority = str(input("what is the priority?")).strip() # ask for priority
156
157      return task, priority # update these variables to be used in the main code
```

figure 2: first entry for assignment

This function was one of two functions to be called when the user selected 1. Between these two functions, a new task and priority will be added to the list. In figure 2, I simply set two different variables equal to what the user inputs. The return function at the bottom of the function ends this function, and at the same time updates the variables to be used in the rest of the program. The figure below shows this function being called, and the variables being set.

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

figure 3: main body calling functions.

First line 184 was called, where task and priority were set. Line 185 now calls a new function, where it will update the variable table_lst. There are three arguments being passed as well, to

be used in this function.

```
1 usage
@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 to add more data
    :return: (list) of dictionary rows
    """
    row = {"Task": str(task).strip(), "Priority": str(prio
    # TODO: Add Code Here!
    table_lst.append(row) # add dictionary to the list

    return table_lst # update the list
```

figure 4: next function call

Here, task, priority, and list_of_rows will be used as variables in the code. The information in the new dictionary is stripped, and added to the table, that is storing the data temporarily. table_list is then updated to be used in the main body. We have now updated the list with new data.

Next we will remove data.

```
188 elif choice_str == '2':  # Remove an existing Task
189     task = IO.input_task_to_remove()
190     table_lst = Processor.remove_data_from_list(task=task, lis
191     continue  # to show the menu
```

figure 5: remove data

Again, line 189 is setting 'task' equal to some value, a value which is established in the function IO.input_task_to_remove().

```
159     @staticmethod
160     def input_task_to_remove():
161         """  Gets the task name to be removed from the list
162
163         :return: (string) with task
164         """
165         pass  # TODO: Add Code Here!
166         task = str(input("which task would you like to remove?")) # prompt user for
167         return task # update the task variable to be used in the main code.
```

figure 6: get task name to remove

Here we ask user for a task to remove, and update the variable 'task', to be used on line 190. Once line 190 is run, it takes us to that function.

```
59      def remove_data_from_list(task, list_of_rows):
60          """ Removes data from a list of dictionary rows
61
62          :param task: (string) with name of task:
63          :param list_of_rows: (list) you want filled with file data:
64          :return: (list) of dictionary rows
65          """
66          # TODO: Add Code Here!
67
68          itemRemoved = False  # Use this to verify that the data was found and remov
69          for row in table_lst: # go through the list, one dictionary at a time
70              task1, priority = dict(row).values() # assign key and value names to ea
71              if task1 == task: # if the key name is equal to any key names scanned,
72                  table_lst.remove(row) # remove the first row that meets true in this
73                  itemRemoved = True # set bool value to true, for the next part of co
74
75
76          if itemRemoved == True:
77              print(f"The item '{task}' has been removed") # if item has been removed
78          else:
79              print("I am sorry, i could not find that task")
80
81          return table_lst # returns updated list of dictionaries to main code.
```

figure 7: actually remove data from list

My comments explain how this code is working, but I would like to talk about whether or not it would be appropriate to add the print functions in this part of the code (like mentioned above in the beginning of this report). This section would normally be reserved for processing, but the print function is IO. I think if I was to write this code from scratch, I would include another IO function, that would take care of lines 76 through 81, and call that function after this one runs.

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

figure 8: saving data

Here is the next function to be called, if the user decides to save. I came up with the following:

```
    1 usage
83  @staticmethod
84  def write_data_to_file(file_name, list_of_rows):
85      """ Writes data from a list of dictionary rows to a File
86
87      :param file_name: (string) with name of file:
88      :param list_of_rows: (list) you want filled with file data:
89      :return: (list) of dictionary rows
90      """
91      # TODO: Add Code Here!
92      file_obj = open(file_name, "w") # open file
93      for row in list_of_rows: # scan list one dictionary at a time
94          file_obj.write(row["Task"] + "," + row["Priority"] + "\n") # write values from d
95                                                                     # by comma
96      file_obj.close() # close file
97
98      return table_lst # update this variable
```

figure 9: saving data function

Again, my notes describe what is happening in this code, but I had trouble understanding why I needed to return the table_lst. I ran the code both with and without this line, and it was apparent that I did need it in my code. Then main reason I didn't think that it was needed, was because I wasn't making any changes to that variable. But after looking around, I noticed that this function call was supposed to equal that variable, and the only way it could update was to return it. I am still not entirely sure why this is needed, only that it is.

At the end of the code, is a piece that is self explanetory:

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

figure 10: exiting the program

Conclusion

What I didn't mention in my report, is that I did use the debugging method at one point. I got hung up on a few different issues, and although it did help a bit, I found it was easier to inject print("x") funtions in multiple places, to see how different variables were changing throughout the code. It helped me in troubleshooting why the list was not printing to the file. It ended up

being a syntax error where I was looking at the dictionary using the indexing word "Tasks", when it was supposed to be "Task". I ended up spending an hour trying to figure out why this one didn't work, but fortunately it made me go through all lines of code one at a time, to really understand what was happening in this code. We learned a lot of good information in this section. I am looking forward for more.