Craig Muth

11/20/2022

IT FDN 110 A Au 22: Foundations Of Programming: Python

Assignment 06

https://github.com/cjmuth/IntroToProg-Python-Mod06

TODO List - Read/Write using Functions.

Introduction

The goal of this project is to create a TODO list program, that will use functions to read data from an existing text file, allow the user to view the existing data, make changes, and save the data back to the text file.

A partial program has been provided to start from. All the necessary structure exists, but operational code has been omitted in several places so it will not run as it currently exists. So we will need to map the logic for the existing code, identify where it is lacking, and develop the logic and code to make it work.

Designing the program

Examining the starter file, the logic flow looks like this - with areas where code is missing is indicated.

- Data
    - declare variables and constants
- Processing
    - class Processor
        - def read_data_from_file
            - open the file in read mode
            - loop through rows in data file
                - read data from file, splitting into discrete values
                - assign the discrete values to a dictionary
                - add dictionary to list_of_rows
            - when end of rows, close file
            - return list_of_rows to main program
        - def add_data_to_list
            - assign passed values to a dictionary
            - *MISSING CODE*
            - return list_of_rows to main program
        - def remove_data_from_list
            - *MISSING CODE*
            - return list_of_rows to main program
        - def write_data_to_file

- *MISSING CODE*
- return list_of_rows to main program

- Input/Output
  - class IO
    - def output_menu_tasks
      - print menu of options to screen
    - def input_menu_choice
      - get user option
      - return choice to main program
    - def output_current_tasks_in_list
      - loop through dictionaries in list_of_rows
        - print values to screen
    - def input_new_task_and_priority
      - *MISSING CODE*
    - def input_task_to_remove
      - *MISSING CODE*
  - call read_data_from_file
  - while true
    - call output_current_tasks
    - call output_menu_tasks
    - call input_menu_choice
    - if "1"
      - call input_new_task_and_priority
      - call add_data_to_list
        - pass values task, priority, table_lst
    - if "2"
      - call input_task_to_remove
      - call remove_data_from_list
        - pass values task, table_lst
    - if "3"
      - call write_data_to_file
        - pass values, file_name_str, table_lst
    - if "4"
      - exit program

In this example the programmer used classes to collect their data and related functions together (*Classes,* (n.d). Retrieved November 20, 2022, from https://docs.python.org/3/tutorial/classes.html) (External link). In this case, they collected the processing functions together in one class, and input/ output functions in another. This allows a more complete separation of concerns and helps keep the segments of code to smaller pieces where their dedicated operations are easier to follow, but can make it more difficult to follow the working values as they pass through the code. However, it can also make it easier to replace or update sections of codes without impacting the overall flow of the program (as long as the interfaces to other code don't change) - making the extra effort worthwhile over time.

Since both the processing and input/output sections include functions exchanging arguments with main program, instead of addressing the code section by section - we'll pull the related pieces from each section. This will make it easier to understand how the values are passed around for each operation.

Add a new task:

When the user selects option 1, the program runs the following code.

```
# 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
```

Figure 1: Menu option 1. Add a new task

This instructs the computer to first run the function `input_new_task_and_priority` from the IO class and assign the output to the variables `task` and `priority`. Looking at the pseudocode above, we can see this function is one of the areas that is missing code.

```
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!
```

Figure 2: Input new task

Neither the call to the function, nor the function itself, show that values need to be passed to the function - but the call in the main program is assigning values to two variables. From the text in the docstring and the flow we derived from examining the code, the logic for this function should be:

- Get new task description from user
- Get priority of new task from user
- Pass the user given values back to the main program

We use input statements to get values from the user, then the `return` statement is used to pass the values back to the main program (Ramos, Leodanis Pozo (n.d.). *The Python return Statement: Usage and Best Practices,* https://realpython.com/python-return-statement/) (External site).

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

    :return: (string, string) with task and priority
    """
    # TODO: Add Code Here!
    # Get new task description from user
    task = input('What is the new task? ')
    # Get priority of new task from user
    priority = input('What is the priority? ')
    # Pass the user given values back to the main program
    return task, priority
```

Figure 3: Updated Input new task

Now that we have values for the new task, the code calls the function `add_data_to_list`, from the processor class, this time we are passing the values from the previous function, along with the existing value of variable `table_list`, and assigns the results back into the `table_list` variable. Looking at the function, we see it is designed with three parameters (`task`, `priority`, and `list_of_rows`) to accept the values passed from the calling function (`task`, `priority`, and `table_list`)

```
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!
    return list_of_rows
```

Figure 4: Add data to list

The code is making use of the first two values by assigning them to a dictionary, does not make use of if, then passes the value of `list_of_rows` (and control) back to the main program. In order to add the dictionary to the list the logic should be updated to something like this:

- Receive values from calling code
- Assign `task` and `priority` to a dictionary
- Add the dictionary to `list_of_rows`
- Return `list_of_rows` to main program

This should only require one line of code to append the dictionary to the list before the `return` statement.

```python
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)
    # Pass the updated task list back to the main program
    return list_of_rows
```

Figure 5: Updated Add data to list

Remove an existing task:

Selection of option 2 runs the following code.

```python
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
```

Figure 6: Menu option 2. Remove an existing task

This time the computer calls the `input_task_to_remove` and assigns the value to `task`.

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

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

Figure 7: Input task to remove

Again, looking at the docstring and derived flow - the logic for this function should be:

- Get description of task to be removed
- Pass the task description back to the main program

Which can be done with only two lines of code.

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

    :return: (string) with task
    """
    pass  # TODO: Add Code Here!
    # Get description of task to be removed
    task = input('\nWhich task do you want to remove? ')
    # Pass the task description back to the main program
    return task
```

Figure 8: Updated Input task to remove

Similar to the way a task was added, the description in `task`, as well as the `table_list` are passed to the `remove_data_from_list` and updated value assigned to `table_list`. We're assuming that each task will be unique and only have one priority assigned, so we only need to pass one value to identify the item to remove.

```
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!
    return list_of_rows
```

Figure 9: Remove data from list

Currently the function is taking no action with the received data before passing back the to main program. In order to remove a task, the function should flow like this:

- Receive values from calling code
- Iterate through list looking for matching task description
    - Remove the task
- Pass the updated task list back to the main program

```
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!
    # Iterate through list looking for matching task description
    for row in list_of_rows:
        if row["Task"] == task:
            # Remove the task
            list_of_rows.remove(row)
    # Pass the updated task list back to the main program
    return list_of_rows
```

Figure 10: Updated Remove data from list

Save data to file:

The final piece of missing code is to save the data to a file. From the existing code we can see this will call the `write_data_to_file` function and pass the `file_name_str` and `table_list`. Although it assigns the results to table_list, the code doesn't appear to make further use of it. This assignment may not be necessary - but will be left in because it won't adversely affect how the program runs, and the original programmer may have plans to make use of it in future updates.

There also appears to be an extra space between the class and function names, which should cause and error when the program runs - we will fix that, but not show the updated code here.

```
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
```

Figure 11: Menu option 3. Save Data to a File

As written, the function does nothing but receive two values and pass one back to the main program.

```
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!
    return list_of_rows
```

Figure 12: Write data to file

- Receive values from the calling code
- Open the file in write mode
- Loop through dictionaries in the table
    - Extract values, concatenate as string, write to text file
- Close text file
- Pass the task list to the main program

```python
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!
    # Open the file in write mode
    objFile = open(file_name, 'w')
    # Loop through dictionaries in table list
    for row in list_of_rows:
        # Extract values, concatenate as string, write to text file
        objFile.write(row['Task'] + ',' + row['Priority'] + '\n')
    # Close text file
    objFile.close()
    # Is this return necessary?
    return list_of_rows
```

Figure 13: Updated Write data to file

From the pseudocode we see this was the last section that was missing code in the starter file - so we should be able to run the program with no further changes

Running the program

Executing the program in Pycharm:

When the program opens it displays the existing list of tasks and the menu of options to the user. As the code for this was included in the starter file it doesn't provide any information on our code, but does verify that the data source exists and we are able to read from it. By selecting the option to exit, we verify that the given code works correctly - which makes troubleshooting easier because any errors or exceptions should be a result of the new code.

```
******* The current tasks ToDo are: *******
Fold laundry (Low)
Pet the cat (Very 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] - 4

Goodbye!

Process finished with exit code 0
```

Figure 14: Testing the functions of the starter program

As the program will be passing values back and forth as it runs, we'll need a way to track the values to ensure the correct values are assigned to each variable as it goes. In order to do this we will add breakpoints to the code and run the Pycharm debugger so the program will halt and allow us to check the variables before proceeding.

Line breakpoints can be inserted in Pycharm by clicking in the “gutter" between the code and the line number (Breakpoints (August 5, 2022), retrieved November 20, 2022 from https://www.jetbrains.com/help/pycharm/using-breakpoints.html) (External link).

```
176
177         # Step 4 - Process user's menu choice
178         if choice_str.strip() == '1':  # Add a new Task
179             task, priority = IO.input_new_task_and_priority()
180             table_lst = Processor.add_data_to_list(task=task, priority=priority, list_of_rows=table_lst)
181             continue  # to show the menu
182
```

Figure 15: Adding breakpoints to the add data functions

Breakpoints will be inserted at all points where our new code will be passing argument values to of from a function - though not all will be shown here.

We launch the program by selecting Debug from the Run menu. Initially it looks almost exactly the same as the previous screen.

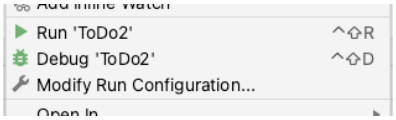


Figure 16: Launching in debug mode

```
******* The current tasks ToDo are: *******
Fold laundry (low)
Pet the cat (Very 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 17: Running in debug mode

If we enter 1 - the program pauses, highlights the line that it’s about to execute, and displays the list of variables in use and the associated values. We can see that it identified the user input correctly, the dictionary `row_dic` is empty and ready for a new value, and `table_list` contains the two tasks that were loaded from the text file.

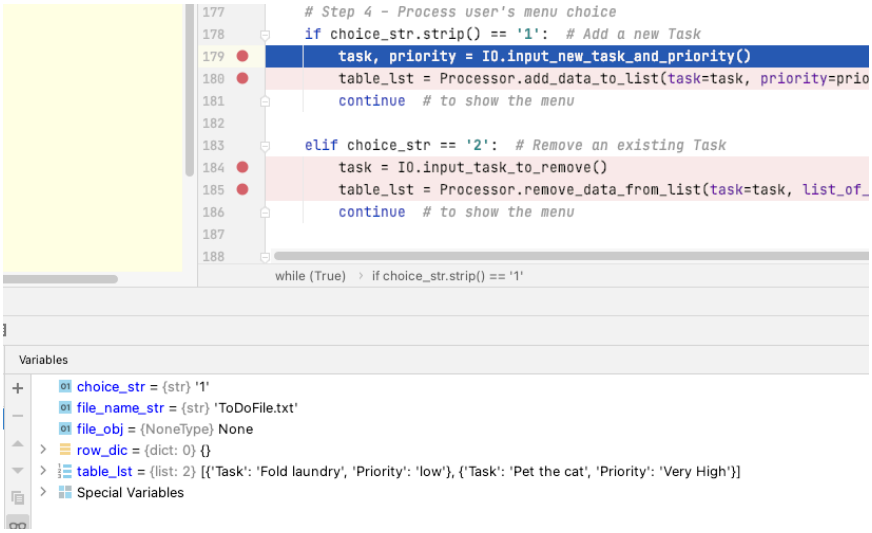


Figure 18: Paused at a breakpoint

If we click the resume button, the program continues from where it left off - we enter values for a new task, and it runs until it reaches the next breakpoint.

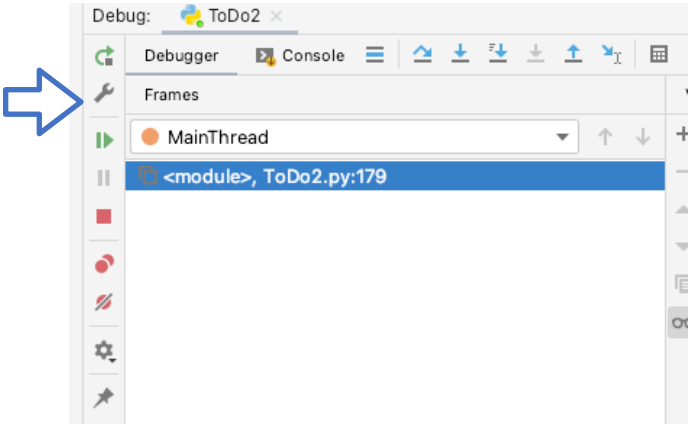


Figure 19: Resume running in Debug mode

```
        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 new task? >? Feed the cat
What is the priority? >? High

>>>
```

Figure 20: Testing the add data functions

Figure 21: Return the user input

We can see the values for task and priority have been captured, and are being passed back to the main program as intended. Checking one more breakpoint, where the program returns from the `add_data_to_list` function - we see all the correct values have been passed in, a new dictionary (`row`) has been created and added to `list_of_rows`.

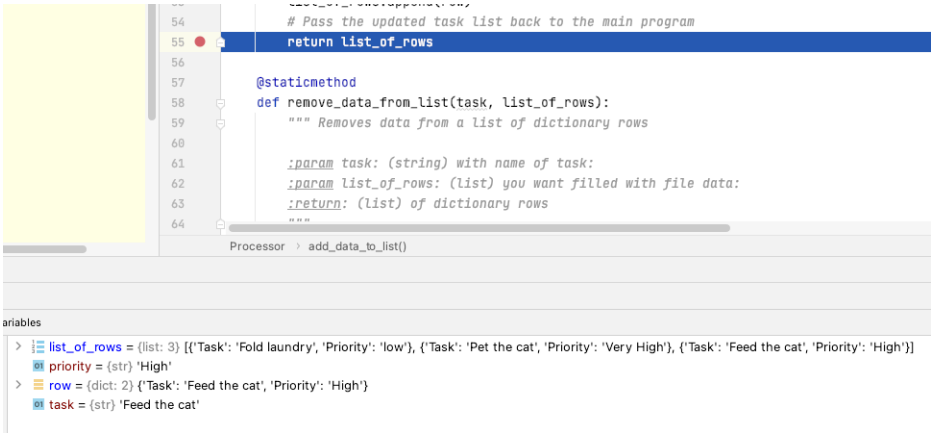


Figure 22: Return from the add data function

Since the program appears to be running correctly, we won't look at any more Debugging screens unless we encounter an error. Testing the rest of the functions we see the following screens.

```
Which option would you like to perform? [1 to 4] - 2


Which task do you want to remove? Fold laundry
******* The current tasks ToDo are: *******
Pet the cat (Very High)
Feed the cat (high)
*****************************************
```

Figure 23: Testing the data removal functions

```
Which option would you like to perform? [1 to 4] - 3

Data Saved!
******* The current tasks ToDo are: *******
Pet the cat (Very High)
Feed the cat (high)
*****************************************
```

Figure 24: Testing the save functions

Finally we exit the program and check the contents of the text file to ensure the changes were saved.

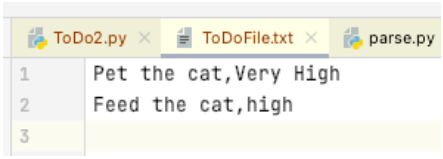
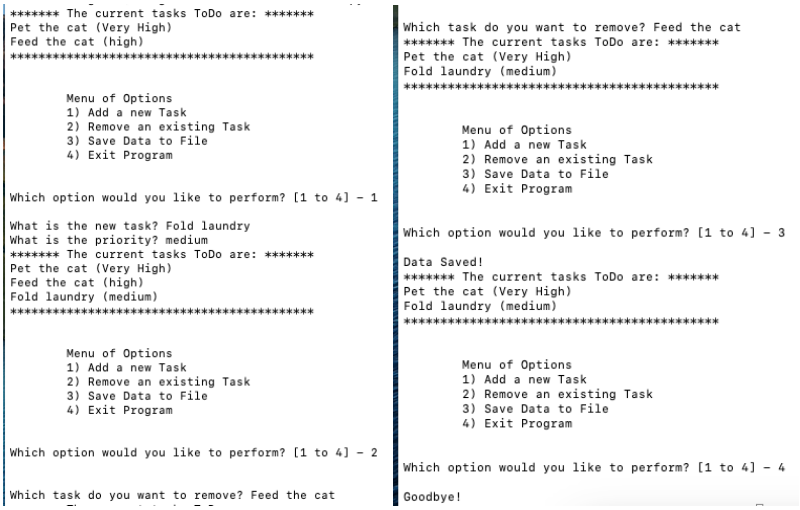


Figure 25: Checking the contents of the text file

Executing in a Terminal window:

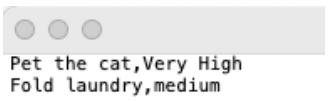
Running the code in a terminal window, we see the following interactions:



```
******* The current tasks ToDo are: *******
Pet the cat (Very High)
Feed the cat (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 new task? Fold laundry
What is the priority? medium
******* The current tasks ToDo are: *******
Pet the cat (Very High)
Feed the cat (high)
Fold laundry (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

Which task do you want to remove? Feed the cat
******* The current tasks ToDo are: *******
Pet the cat (Very High)
Fold laundry (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] - 3

Data Saved!
******* The current tasks ToDo are: *******
Pet the cat (Very High)
Fold laundry (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] - 4

Goodbye!
```

Figure 26: Running in a terminal window

Checking the contents of the text file, we see the program is working as intended.



```
Pet the cat,Very High
Fold laundry,medium
```

Figure 27: Re-checking the contents of the text file

Summary

In this project we created a program to interact with text file - using functions to read from a file, modify the data, and write it back to the source data file. Given a partial program to start with, we broke it down into pseudocode to help understand the initial logic flow and identify the areas that needed additional code in order to complete the program.

After identifying the incomplete functions and reviewing their inputs and outputs, we determined the logic needed for the necessary transformations - then created and tested the code. By using the debugger in Pycharm, we were able to verify the values of the arguments passed to the new functions, and verify the return values as the program progressed - rather than simply executing the code and waiting for the final result to determine if it functioned correctly.