Hannah Clayton

February 15th, 2021

Foundations of Programming: Python

Assignment 05

[GitHub URL](https://github.com/hnolte-grad/Intro-to-Prog-Python/blob/main/Assignment05_Main.py)

05\_To Do List/Working with Dictionaries

I. Introduction

This document serves as a demonstration of my experience working through this week’s assignment module. In the ‘module objectives’ section, I highlight the intended goals for this week’s reading and instructional video as given by Professor Root, and in my own words communicate my understanding of the objectives. In the ‘assignment work-through’ section I show how I went about approaching the module’s assignment, and any issues or epiphanies I encountered in doing so. The overall intent of this document is to serve as both a reference for myself and others in the future and demonstrate my competency in the module to Professor Root.

II. Module Objectives

1. Objectives

• What is the difference between a List and a Dictionary?

• What is the difference between an Index and a Key?

• How do you read data from a file into a List?

• How do you read data from a file into a Dictionary?

• What is the programming pattern called “Separations of Concerns?”

• How would you use a function to organize your code?

• Why is a script template useful?

• Why is error handling using Try-Except recommend?

• What is GitHub, and why is it used?

1. Objective Summary

This week’s course module introduced us to dictionaries, their functions and methods, and the similarities and differences they share with lists. We also touched on using the online code repository, *GitHub*, to back up our files, collaborate, and get feedback on our work from fellow classmates and other professionals in the future. Lastly, we discussed ways in which an individual can make their coding process more efficient and user-friendly.

*Dictionaries* vary from lists primarily in the use of keys and indexes, respectively, to handle data. Dictionaries operate using ‘keys’ and ‘values’ like seen in the following line of code: **dictionary = {"ID":"1","Name":"Hannah", "Weight":"125"}.** Dictionaries are established using a braces ‘**{}**’ operator rather than brackets **‘[]’** as lists do. Furthermore, dictionary ‘*keys*’ are composed of character subscripts instead of numerical indexes. When multiple dictionaries are put into a list they create a two-dimensional table where it can be helpful to think of the dictionaries’ keys as column headings and of each dictionary as a single ‘row’ of data. Image 2.1 helps visualize the concept of dictionaries as a table. In programming language Image 2.1 and its data would be represented by the following lines of code:

**Image 2.1 Dictionaries as a table**

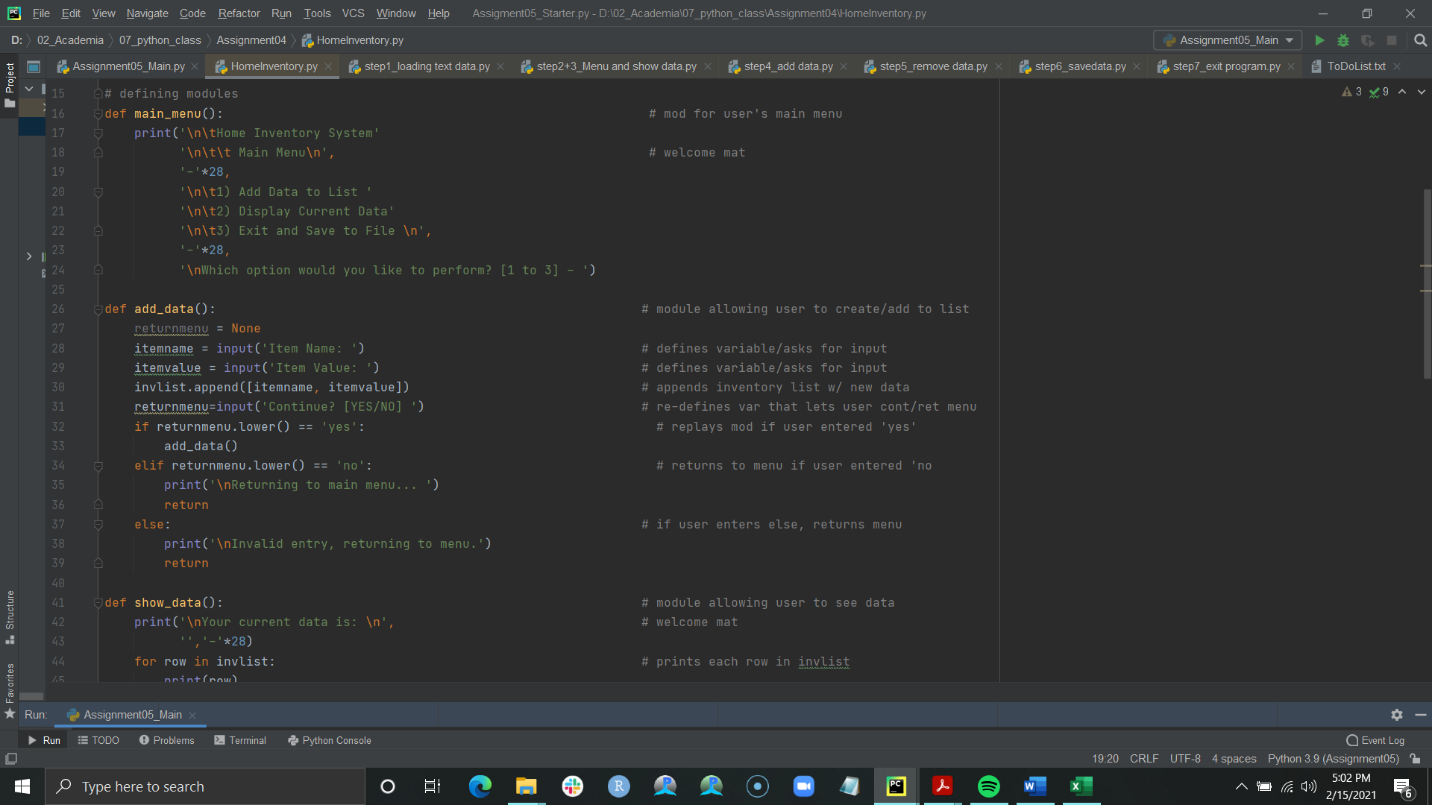
**dicRow1= {‘Fish ID’: 1, ‘Fish Name’: ‘OJ Swimson’, ‘Breed’: ‘Neon Tetra’, ‘Color’: ‘Orange’}**

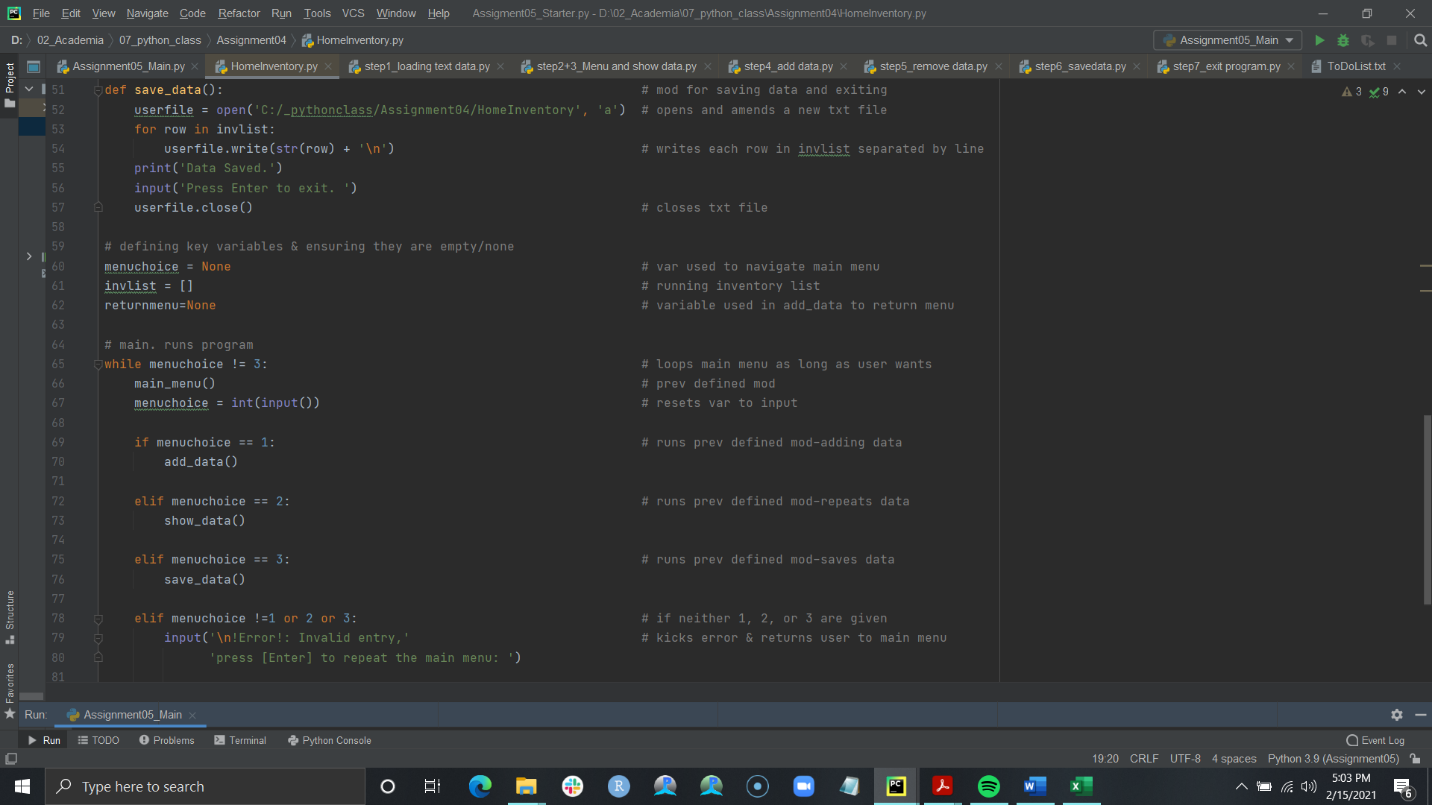
**dicRow2= {‘Fish ID’: 2, ‘Fish Name’: ‘S!nk’, ‘Breed’: ‘Neon Tetra’, ‘Color’: ‘Pink’}**

**lstTable= [dicRow1, dicRow2]**

 Lists and dictionaries have different methods and ways of achieving the same task, such as loading data from a text file. Lists are often saved to text files, and therefore loading a list from a text file into an active list is fairly easy, assuming the text file data is actual a list and not a string type. Image 2.2 first shows loading data from a text file into an active list, which requires indicating which character to separate the data with (line #10) and looping through the data in the file. Loading into a dictionary follows the same basic concept, but with a few more steps. You first must load the data from the file into an active list (line #15), and then create your dictionary rows using the index values in the list (line #16).

**Image 2.2 Loading data in list v. dictionary**

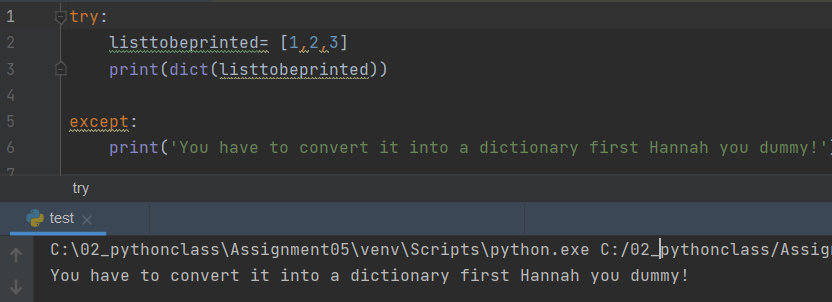
 As we have reached the half-way mark in our course our code has inevitably gotten longer and more complex. There are a few widely accepted coding habits and patterns necessary for quicker and more efficient coding, such as the use of pseudo-code as recently covered. When beginning work on a script formatting your code using the ‘*Separation of Concerns*’ design principle helps keep your code organized. The majority of scripts and code can be divided into three sections: *Data*, or where you identify key variables; *Processing*, where the bulk of your computations is done such as calculations or data conversion*; Presentation*, where you get input or communicate output.

Using the Separation of Concerns design principle can be difficult as there are many times you need to ask for user input to perform computational tasks, which requires a mixing of the two sections Processing and Presentation. Declaring *functions* make this easier, as you can group lines of code together by an identified variable, and later reference that variable/those statements to run that line of code. In last week’s assignment I utilized both functions and separations of concern, despite not understanding entirely what these topics were formally called and referred to them as ‘modules’. In Image 2.4 you can see that I broke each menu choice into a different function using the ‘def’ operator and assigning a variable. Image 2.5 shows how I referenced each different function under its correlating conditional statement, making for a cleaner loop sequence.

**Image 2.3 Defining functions (Assignment 04 Example)**

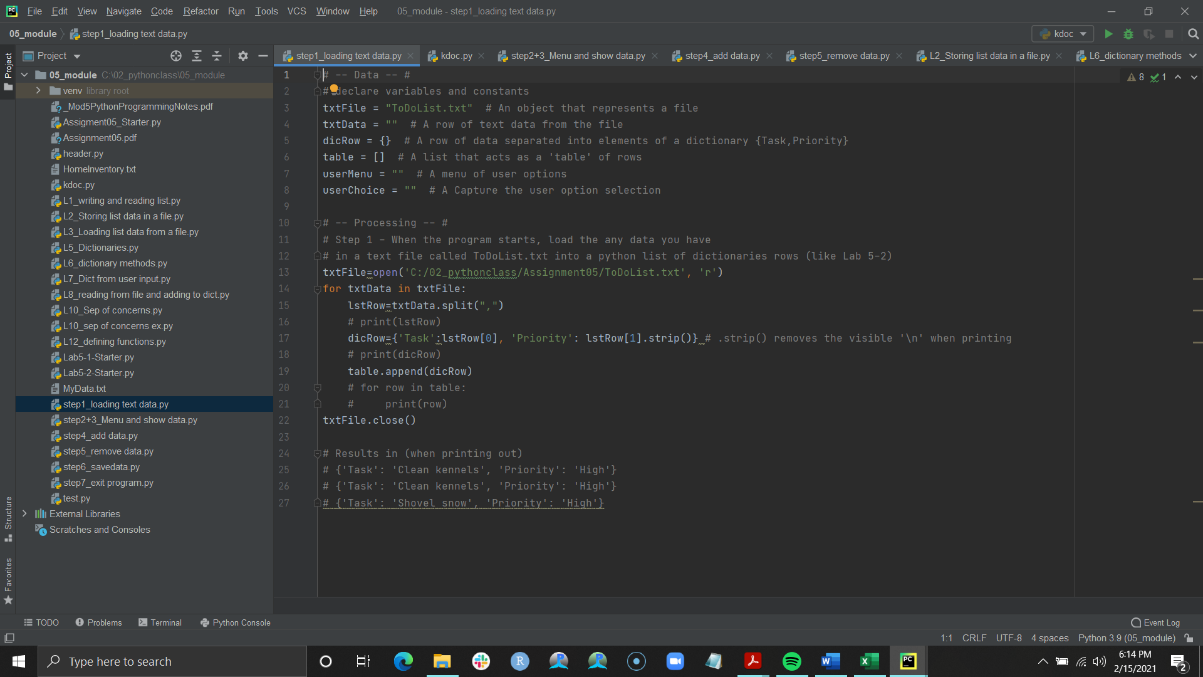
**Image 2.5 Referencing functions in a loop (Assignment 04 Example)**

Typing out the same kinds of statements or codes lose their novelty over time and it becomes useful to utilize *script templates* for common functions or tasks you utilize. A good example of a script template is the text file I have saved containing a blank header I copy and paste into each script I create. As we learned this week though, templates are so frequently used by programmers that most IDEs, like Pycharm, allow you to save a file as a template.

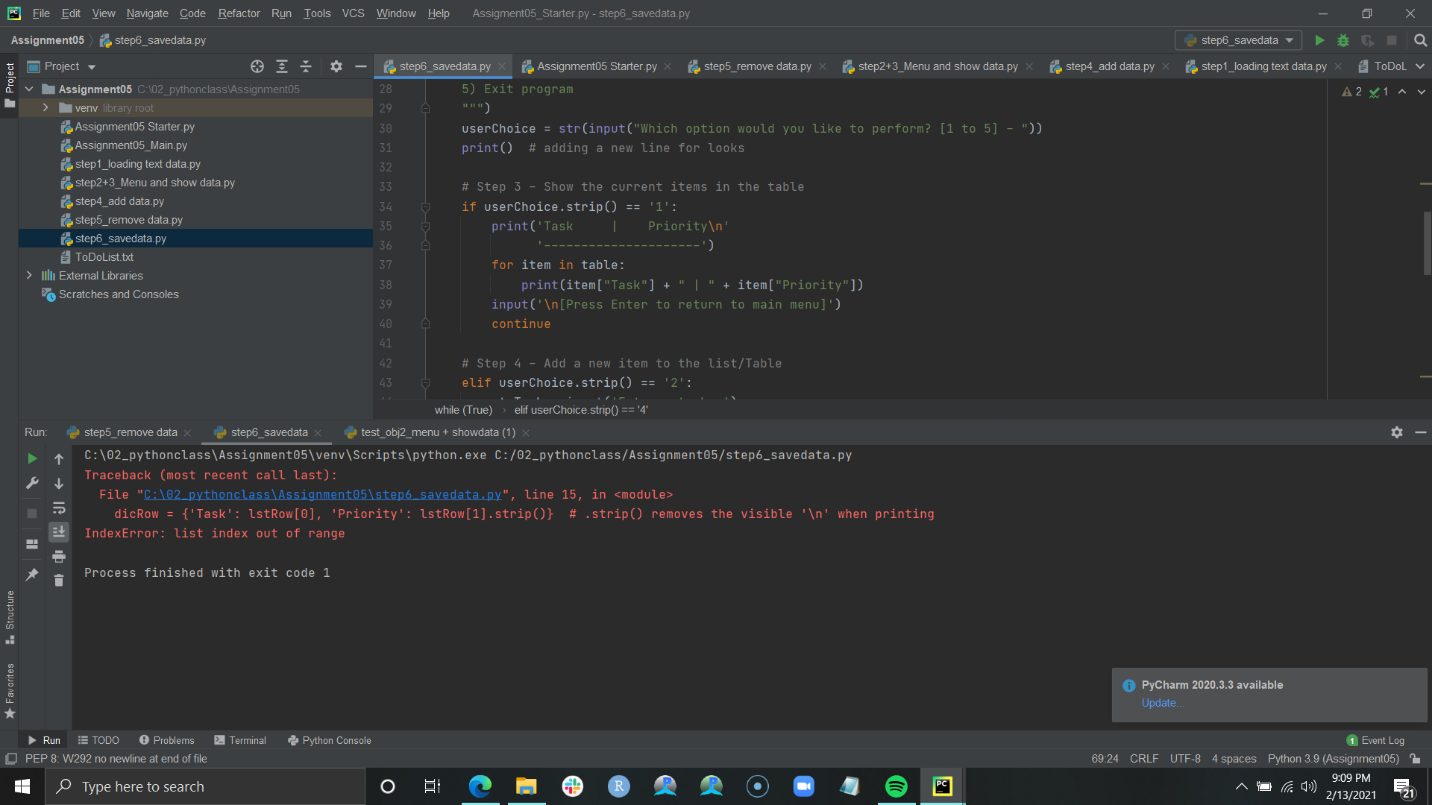
 The last tool in creating clean and efficient code we covered this week was the use of try-except statements. *Try-except blocks* allow you to change the way your program reacts when it encounters an error. This can be useful as the majority of individuals interacting with programs have little to no understanding of computer programming or languages and may not understand the error message given to them. To set up a try-except block you first trap the program in an error using the ‘try’ function, and then instruct it on how to respond to that error. An example of what that code looks like can be seen in Image 2.6.

**Image 2.6 ‘Hannah still doesn’t understand type conversion’ error message**

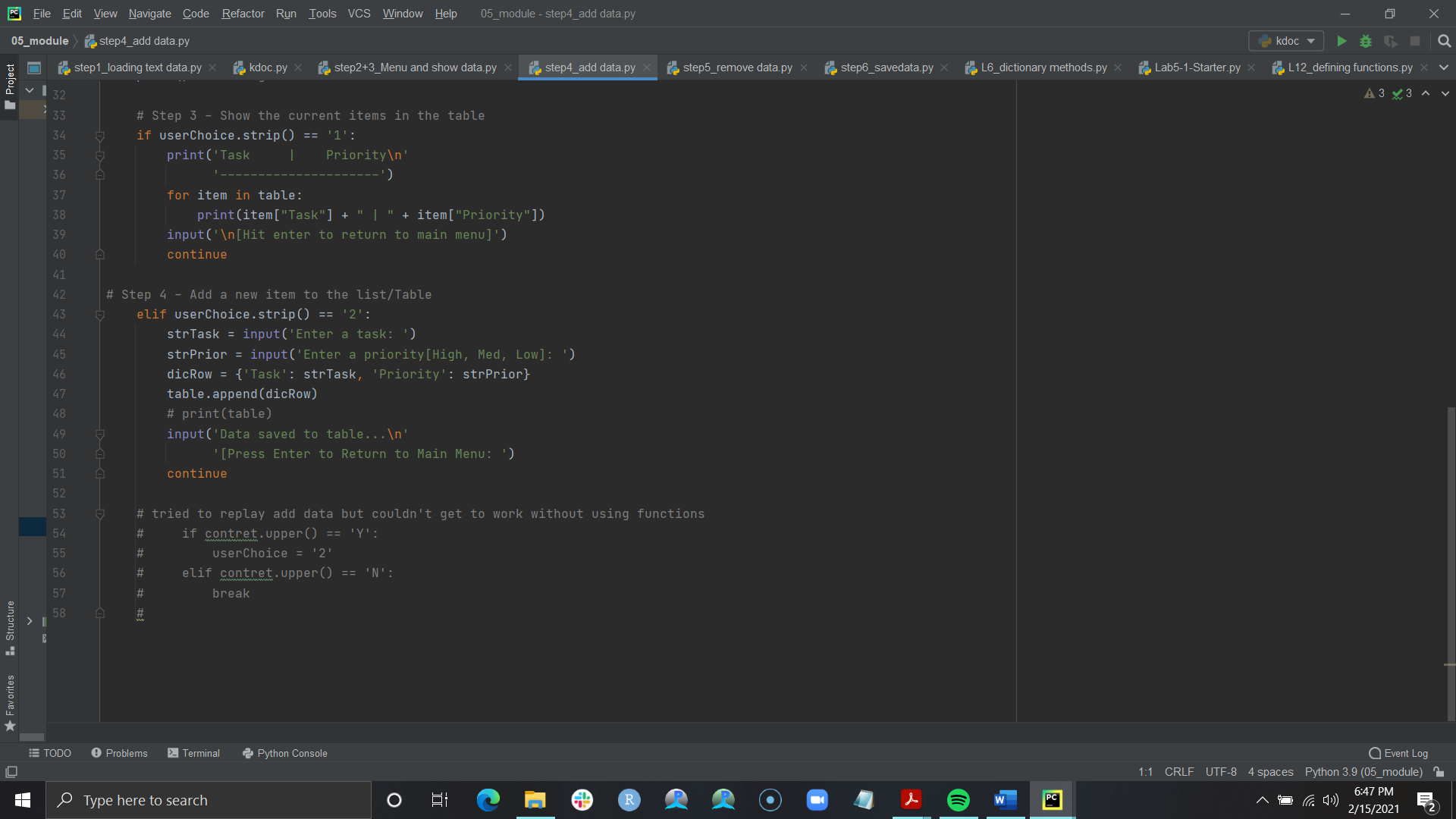
III. Assignment Work-through

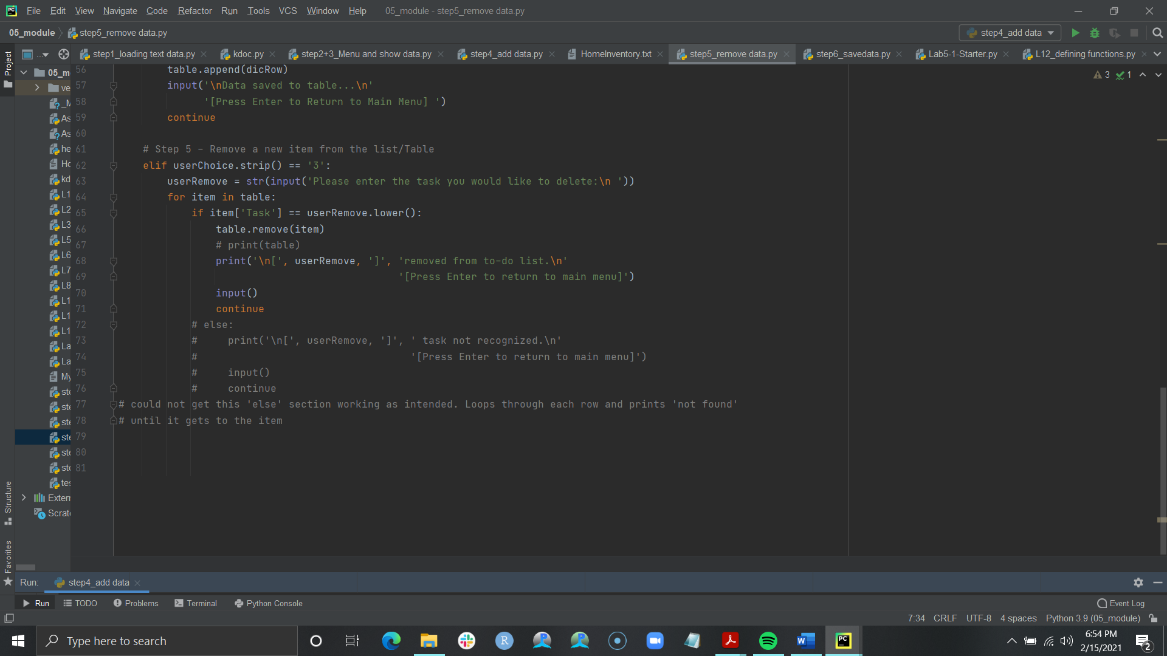
This week’s assignment required us to add code to a script provided to us by Professor Root and add code to accomplish tasks such as saving dictionary data to a text file, loading data from a file into a dictionary, and adding or removing items from a table. I found this assignment a little harder to work through than last week’s primarily because last week I used functions to better help me flow through creating the code. Additionally, I found that while having a starter code should have cut down the time I spent writing, at this stage in my programming abilities it takes me just as long to truly comprehend each part of someone else’s code as it does for me to sit down and write it all out myself. Starting out with that much code was a little overwhelming to look at, so I took an idea from GitHub and broke it all down. I took the starter code and created a separate script file within the Assignment 05 project for each step requiring me to enter code. I then created an ‘Assignment05\_Main’ script where I would paste each step once I was finished with it. After I completed the code, I moved those step scripts to my Module 05 project since I did not want them uploaded with my final script, but Image 3.1 shows an example of those separate script steps. Overall, I think breaking it down this way was helpful since I could not use functions and I could insert comments, test things out, etc. and not have to delete them so they were not in my final script. Additionally, when I did run into errors, I could go back to that step’s script to troubleshoot and test without so much clutter.

**Image 3.1: Separation of steps**

 Most of writing and executing this code went easily once I found a good workflow, but I did run into one error that took me a while to parse out. This error occurred after I had completed all the code except exiting the program. Initially, I had no issues loading data from a text file, however, when I changed ‘save data’ formatting and saved data, I would get an index error when starting the program again (Image 3.2). The interesting part was that the program would load some of the data (from before I altered the ‘save’ code), and then throw the index error. So, I knew there had to be a disparity between how I was saving the data and how I was asking the program to load it. It took a lot of testing and typing, but eventually I noticed that when writing my save code I indicated that I wanted data to be separated by ‘ | ‘, but when asking the program to load that data from the file, I indicated I wanted the rows split by “,” as outlined in the starter script. Once I changed the split character to match (and cleared the text document contents), I was able to loop through and load all the data from the text file.

**Image 3.2: Index Error**

 While these are not necessarily errors, there were some additional tasks that I had hoped to include in my script that I could not quite figure out. I wanted the user to be able to continue adding data if they chose, like I did with last week’s assignment, but found that without using functions ‘break’ did not work properly as I could not recall the main menu and ‘continue’ did not want to work in a nested statement (Image 3.3).

 I also wanted to enter a prompt in the ‘remove data’ section for if the user entered a task to remove that was not listed, but because of the nature of the loop, it would throw an error as it went through each row in the ‘if’ condition, and then move on to the else condition. I thought I could remedy this by switching the two, but that did not work as intended either.

**Image 3.4: Data not found**

**Image 3.3: Continue adding data**

My final code result can be seen in the text below this paragraph. Though I am satisfied with it, I am looking forward to digging deeper into functions and other tools for making code cleaner. Please note that because of formatting differences between Pycharm and Word I do not recommend copying and pasting the code below to run it, as I make minor changes to it in Word for viewing purposes.

# ------------------------------------------------------------------------ #  
# Title: Assignment 05  
# Description: Working with Dictionaries and Files  
# 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  
# Hannah Clayton,12Feb21, Added code to initialize assignment 5  
# Hannah Clayton, 13Feb21, Broke down each segment into different files  
# for easier comprehension, input code for steps  
# 1-5  
# Hannah Clayton, 14 Feb 21, Resolved error 'index value out of range' that  
# that was caused by incorrect typing in processing  
# section, finished code.  
# ------------------------------------------------------------------------ #  
# -- Data --#  
txtFile = "ToDoList.txt" # An object that represents a file  
txtData = "" # A row of text data from the file  
dicRow = {} # A row of data separated into elements of a dictionary   
table = [] # A list that acts as a 'table' of rows  
userMenu = "" # A menu of user options  
userChoice = "" # A Capture the user option selection

# -- Processing -- #  
# load data from text file when starting  
txtFile = open('C:/02\_pythonclass/Assignment05/ToDoList.txt', 'r')  
for row in txtFile:  
 lstRow = row.split(" | ")  
 dicRow = {'Task': lstRow[0], 'Priority': lstRow[1].strip()}   
 table.append(dicRow)  
txtFile.close()  
  
# -- Input/Output -- #  
# User menu  
while (True):  
 print("""  
 Menu of Options  
 1) Show current data  
 2) Add a new item  
 3) Remove an existing item  
 4) Save data to file  
 5) Exit program  
 """)  
 userChoice = str(input("Which option would you like to perform? [1 to 5] - "))  
 print() # adding a new line for looks  
  
 # Step 3 - Show the current items in the table  
 if userChoice.strip() == '1':  
 print('Task | Priority')  
 print('-' \* 21)  
 for item in table:  
 print(item["Task"] + " | " + item["Priority"])  
 input('\n[Press enter to return to main menu]')  
 continue  
  
 # Step 4 - Add a new item to the list/Table  
 elif userChoice.strip() == '2':  
 strTask = input('Enter a task: ')  
 strPrior = input('Enter a priority[High, Med, Low]: ')  
 dicRow = {'Task': strTask.lower(), 'Priority': strPrior.lower()}  
 table.append(dicRow)  
 print('\nData saved to table...\n'  
 '[Press Enter to Return to Main Menu] ')  
 continue  
  
 # Step 5 - Remove a new item from the list/Table  
 elif userChoice.strip() == '3':  
 userRemove = str(input('Please enter the task you would like to delete:\n '))  
 for item in table:  
 if item['Task'] == userRemove.lower():  
 table.remove(item)  
 print('\n[', userRemove, ']', 'removed from to-do list.\n'  
 '[Press Enter to return to main menu]')  
 input()  
 continue

# Step 6 - Save tasks to the ToDoList.txt file  
 elif userChoice.strip() == '4':  
 txtFile = open('ToDoList.txt', 'w')  
 for item in table:  
 txtFile.write(item['Task'] + ' | ' + item['Priority'] + '\n')  
 txtFile.close()  
 print('-'\*21 + '\n')  
 print('Data saved to file.\n')  
 print('-'\*21 + '\n')  
 input('[Press Enter to return to main menu]')  
 continue  
  
 # Step 7 - Exit program  
 elif userChoice.strip() == '5':  
 check = input('Are you sure you want to exit? [Y/N]')  
 if check.lower() == 'y':  
 print('Goodbye!')  
 break  
 elif check.lower() == 'n':  
 continue