Daniel Castro

August 9th, 2023

IT FDN 110 A Su 23: Foundations Of Programming: Python

Assignment 05

Lists and Dictionaries

Introduction

In Assignment 5, we will explore working with Lists and Dictionaries in Python, alongside the process of posting work to GitHub. This assignment aims to enhance our script management skills by introducing script templates and basic error handling. Throughout the activities, we will learn to manipulate and structure data using lists and dictionaries, not only writing data to files but also reading data from them as if it were stored in memory. Additionally, we will initiate our use of GitHub for uploading files. The practical aspect involves modifying a provided script that manages a "ToDo list" by adding, deleting, and saving data in a two-dimensional list, with each task and priority represented as a dictionary "row" of data.

Lists Methods

Alongside basic operations like adding and removing elements, Python has a variety of built-in methods to manipulate collections (in this case, lists) more efficiently. These methods enable tasks like sorting and finding specific elements, amongst others. To explore these methods further, one can always refer to the official Python documentation on list methods and other data structures (Python Documentation: Data Structures, https://docs.python.org/3/tutorial/datastructures.html#more-on-lists, 2012)(External Link). Table 1 shows some selected list methods from the course textbook (M. Dawson, Python® Programming for the Absolute Beginner, 3rd Edition, 2010).

TABLE 5.1 SELECTED LIST METHODS

Method	Description
`append(value)`	Adds *value* to end of a list.
`sort()`	Sorts the elements, smallest value first. Optionally, you can pass a Boolean value to the parameter `reverse`. If you pass `True`, the list will be sorted with the largest value first.
`reverse()`	Reverses the order of a list.
`count(value)`	Returns the number of occurrences of *value*.
`index(value)`	Returns the first position number of where *value* occurs.
`insert(i, value)`	Inserts *value* at position *i*.
`pop([i])`	Returns value at position *i* and removes value from the list. Providing the position number *i* is optional. Without it, the last element in the list is removed and returned.
`remove(value)`	Removes the first occurrence of *value* from the list.

Table 1. Selected List Methods. (M. Dawson, 2010)

Nested Sequences

It's possible to create nested lists or tuples as per usual, by typing each element followed by a comma. The difference with nested sequences is that they include entire lists, tuples or even dictionaries as elements. E.g.,

```
nested = ["first", ("second", "third"), ["fourth", "fifth", "sixth"]]
print(nested)
```

Result:

```
['first', ('second', 'third'), ['fourth', 'fifth', 'sixth']]
```

Figure 1. Example of nested sequences. (M. Dawson, 2010)

To access an element from a nested sequence, one can use multiple indexing to get directly to a nested element. For example, by supplying two indices with `nested[1][0]`, it tells the computer to go to the element from "nested" at position 1 (which is `("second", "third")`) and then, from that, to get the element at position 0 (which is `"second"`).

Another useful tool is unpacking sequences, which we already previewed in class. NOTE: Just a friendly reminder to use the same number of variables as elements in the sequence, because otherwise it will generate an error (M. Dawson, Python® Programming for the Absolute Beginner, 3rd Edition, 2010).

Using dictionaries

"Dictionaries" are a distinct type of collection, similar to lists, as they are mutable. Each entry in a dictionary consists of a key-value pair enclosed in curly braces and separated by colons, like this: {key: value}. However, there's a crucial distinction when working with dictionaries compared to tuples or lists. Dictionaries lack "position-based" indexing. To retrieve a value from a dictionary, you use a key instead. The get() method provides an alternative for accessing values via keys, but if a key isn't present in the dictionary, the method returns `None`. (M. Dawson, Python® Programming for the Absolute Beginner, 3rd Edition, 2010). Table 2 shows some selected dictionary methods from the course textbook.

TABLE 5.2 SELECTED DICTIONARY METHODS

Method	Description
`get(key, [default])`	Returns the value of *key*. If *key* doesn't exist, then the optional *default* is returned. If *key* doesn't exist and *default* isn't specified, then `None` is returned.
`keys()`	Returns a view of all the keys in a dictionary.
`values()`	Returns a view of all the values in a dictionary.
`items()`	Returns a view of all the items in a dictionary. Each item is a two-element tuple, where the first element is a key and the second element is the key's value.

Table 2. Selected Dictionary Methods. (M. Dawson, 2010)

Dictionary Requirements

There are a few things to keep in mind when using dictionaries:

- You can't have multiple items with the same key, just like a real dictionary.
- A key has to be immutable. As such, can be a string, a number, or a tuple.
- Values can be mutable or immutable.

Getting into practice

In the practical part of Assignment 5, we will be working within a provided script that manages a "ToDo list" using Python's Lists and Dictionaries. Our objective is to modify this script to allow users to add, delete, and save tasks and priorities within a two-dimensional List. In it, each task and priority will be represented as a dictionary "row" of data within the List, forming a structured table of tasks.

Creating the script

Starting with the template provided in the "Assignment05_Starter.py" script, I decided to re-order the TODO comments to use them as a form of pseudo-code, guiding my steps during the resolution process. First, I added my information to the comment at the beginning of the script, in the appropriate section of the ChangeLog.

```
# ------------------------------------------------------------------------ #
# Title: Assignment 05
# Description: Working with Dictionaries and Files
#              When the program starts, load each "row" of data
#              in "ToDoList.txt" into a Python Dictionary.
#              Add each dictionary "row" to a python list "table".
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# DanielCastro,8.8.2023,Added code to complete assignment 5
# DanielCastro,8.9.2023, Reformatted the "TO DO" comments and organized the doc
# ------------------------------------------------------------------------ #
```

Then, in the Data Section from the separation of concerns, I verified which variables were being initialized to use them throughout the code. I also placed a TODO comment for this as step 0.

```
# --- DATA --- #
# TODO: Step 0. Declare variables and constants

objFile = "ToDoList.txt"  # An object that represents a file
strData = ""  # A row of text data from the file
dicRow = {}  # A row of data separated into elements of a dictionary {Task,Priority}
lstTable = []  # A list that acts as a 'table' of rows
strMenu = ""  # A menu of user options
strChoice = ""  # A Capture the user option selection
```

Then, moving on to the Processing Section, I decided to use try/except for error handling (in case the text file didn't exist in the first place). It will first try to open the file in read mode and then run through every row of data and unpack the elements on each row into task and priority, while also stripping empty values and splitting them with a ",". This way, it's easier to get the data cleaned to handle it later on if needed.

It will then create a dictionary for each row of data and add them to a list/Table with the append method to then close the text file. However, if it's not able to find the text file in the corresponding

directory, the exception IOError (which can be found on the Table 7.6 from the course textbook) will make sure to pass a statement that print a line to inform the user that there was no initial data found.

```
# --- PROCESSING --- #
# TODO: Step 1. At the start, load any data present in ToDoList.txt into a python list of
dictionaries rows

try:  # Using try/except for error handling if the text file doesn't exist
    rFile = open(objFile, "r")
    for strData in rFile:
        task, priority = strData.strip().split(",")  # Unpacking with the file format as
"Task,Priority"
        dicRow = {"Task": task, "Priority": priority}  # Create a dictionary for each row
        lstTable.append(dicRow)  # Add the dictionary to the list using the append method
    rFile.close()
    print("\n[Data successfully loaded from file]")
except IOError:  # I/O exception when one attempts to open a nonexistent file in read mode
(Table 7.6 from textbook)

    print("\n[No data found. Starting with a blank entry]")
```

Moving on to the Presentation Section, Step 2, which displays the menu of choices, was left as it was originally created in the template.

```
# --- INPUT/OUTPUT --- #
# TODO: Step 2. Display a menu of choices to the user

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
    """)
    strChoice = str(input("Which option would you like to perform? [1 to 5] - "))
    print()  # adding a new line for looks
```

This was followed by the core of this practical assignment, where we had to create the code to handle each user choice or condition.

Option 1: Showing current data.

For this first option, I used an if/else statement to first verify if the length of the list/Table was equal to 0. This way I could verify if there was any information available inside of the text file for starters. Then, the main part of the code that displays the current data will run in the else condition. It will iterate through the list of dictionaries with a for loop and print each row of data as Task, Priority.

```
if strChoice.strip() == '1':
    # TODO: Step 3. Show the current items in the table

    if len(lstTable) == 0:  # Condition in case there's no data in the text file initially.
        print("[No data available]")
    else:
        print("Current Data:")
        for dicRow in lstTable:  # Iterate through the list of dictionaries
            print(f"Task: {dicRow['Task']}, Priority: {dicRow['Priority']}")

    continue
```

Option 2: Adding a new item.

For option 2, the user is prompted to enter a task and next to enter its priority. Then, it creates a new dictionary for the inputs and adds the dictionary to the list. Then, it prints the task that was added and its respective priority, just as some extra information for the user.

```
elif strChoice.strip() == '2':
    # TODO: Step 4. Add a new item to the list/Table

    nTask = input("Enter the task: ")          # New task
    nPriority = input("Enter its priority: ")  # New priority
    dicRow = {"Task": nTask, "Priority": nPriority}    # Create a new dictionary from the
inputs
    lstTable.append(dicRow)  # Add the new dictionary to the list
    print(f"\n[{nTask}] was added to the 'ToDo List' with priority [{nPriority}]")

    continue
```

Option 3: Removing an existing item.

Option 3 is a little bit more complex. I tried thinking of some options first. For starters, I placed an if condition to verify if the length of the list/Table was equal to 0, so that the user is aware of the fact that there's no data available. Then, akin to option 1, I placed the code to remove the item in the else conditional statement. On it, it will first provide information on the tasks that are present in the execution using a for loop that prints every "task" within a row (using the key as index) to then ask the user which task he would like to remove, and finally deleting the row of data from the list/Table, using the remove( ) method. If the task inserted isn't already stored, it will print that said task wasn't found.

NOTE: I also added a flag to verify if the task was indeed removed, and an option that, if the user doesn't select any task to remove and presses Enter instead, it will break the condition and take him to the main menu.

```
elif strChoice.strip() == '3':
    # TODO: Step 5. Remove a new item from the list/Table

    if len(lstTable) == 0:  # Condition in case there's no data to remove yet.
        print("[No data available]")
    else:
        print("Here are the tasks inserted so far:")
        for row in lstTable:
            print(row['Task'])
        removeTask = input("\nEnter the task you want to remove (Press Enter if you changed your mind): ")
        removed = False     # Using a flag to track if the task was removed
        for row in lstTable:
            if row["Task"] == removeTask:
                lstTable.remove(row)  # Using the remove() method to eliminate a row from the list/Table
                removed = True
                print(f"\n[{removeTask}] has been removed from the list.")
                break
            elif row["Task"] == "":   # Elif statement to exit if nothing is going to be removed
                break
            else:
                print(f"[{removeTask}] wasn't found.")

    continue
```

Option 4: Saving data to the File.

In option 4, it just opened the text file in write mode and write down each row of data that is stored in the list/Table as "Task,Priority" and then will just close the file, while printing a text that let's the user know that the data was successfully saved to the text file.

```
elif strChoice.strip() == '4':
    # TODO: Step 6. Save tasks to the ToDoToDoList.txt file

    wFile = open(objFile, "w")     # Open the file in write mode
    for row in lstTable:
        wFile.write(row["Task"] + "," + row["Priority"] + "\n")    # Write each dictionary
row to the file
    wFile.close()
    print("Data saved to ToDoList.txt.")

    continue
```

Option 5: Exiting the program.

Then, for option 5. The program will just print that it's exiting the program and proceed to break and exit it. **NOTE:** To prevent the script from exiting when using the Windows command line, the input from previous assignments was re-used, asking the user to press Enter to end the program and exit.

```
elif strChoice.strip() == '5':
    # TODO: Step 7. Exit program

    print("Exiting the program...")
    input("\n[Press Enter to exit the program]")  # Input to avoid the program to close
Windows Command Line

    break  # and Exit the program

else:   # else statement to make sure the user selects an option in range.
    print("Please select an option from 1 to 5.")
```

Finally, there was another option set up by using the else statement. This one was placed to ensure the user enters values from 1 to 5.

Running the script in PyCharm

Figure 2 shows the program running on PyCharm.

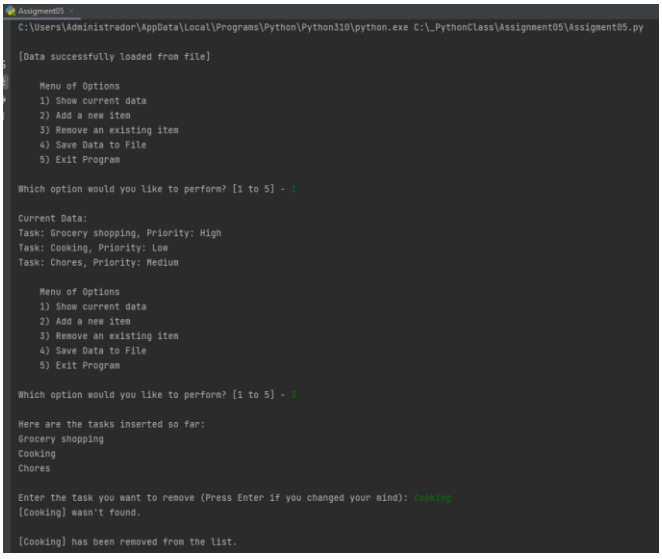

Figure 2. Assignment05.py running with PyCharm.

Running the script in Windows Command Line

To run the script from the Windows Command Line, it's just accessing the directory where the 'Assignment05.py' file is saved and type the name of the Python file. The script will start running (Figure 3).

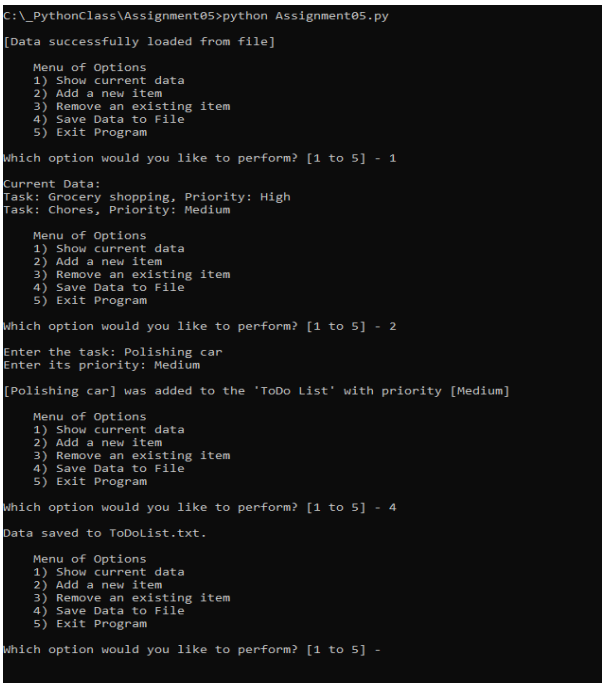

Figure 3 Assignment05.py running in Windows Command Line.

Once we press Option 5 and then press Enter, the code stops running. Figure 4 displays how the data was saved inside the ToDoList.txt file.

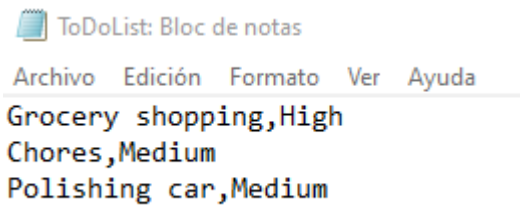

Figure 4. ToDoList.txt file with the saved data.

Summary

In Assignment 5, we explored Python's List and Dictionary concepts and applied them to practical scenarios. We also developed a script that manages a practical "ToDo list", allowing us to add, display, and remove tasks, and read/save data to a text file. So far, this activity has used most of the knowledge acquired throughout the course, since the scripts are starting to look more and more complex, but we use the same basic operators and statements learned before. We also focus on handling data using lists of dictionaries, providing a structured format for presenting information, and also loading data from a file, using menus for user interactions, and applying fundamental list and dictionary methods as well as the concepts of separation of concerns and custom error handling.