Slide Deck presentation Terminal Application

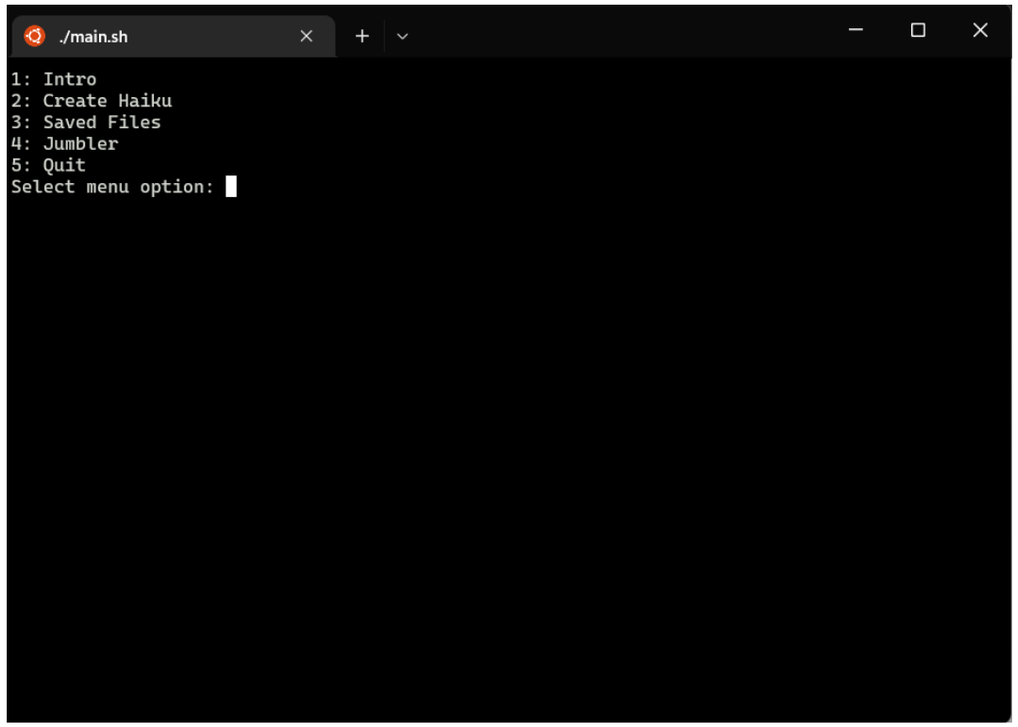
By Liam Massey

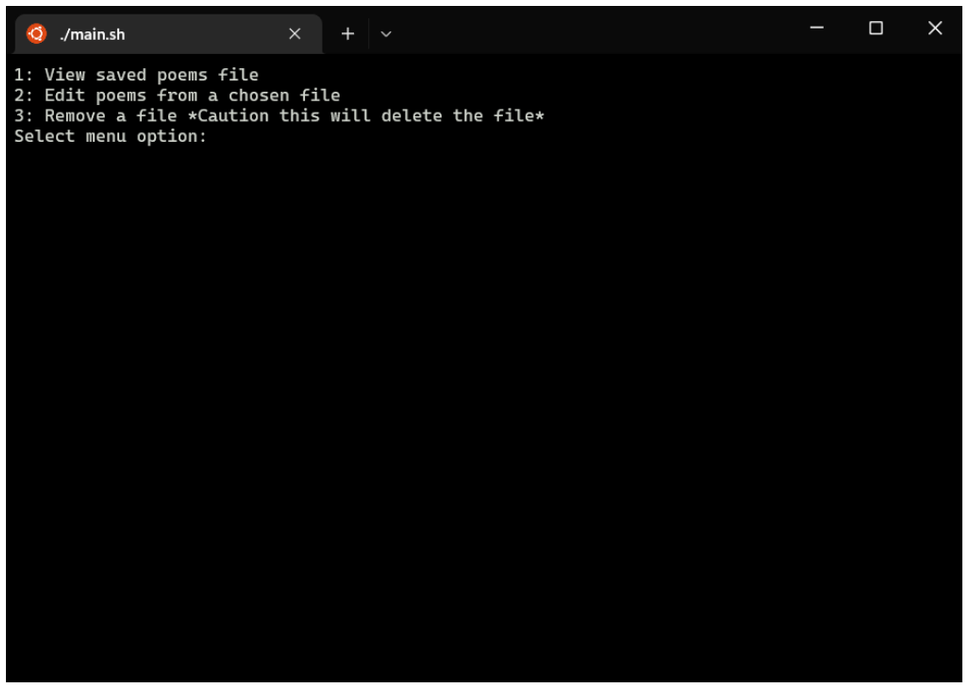
Contents

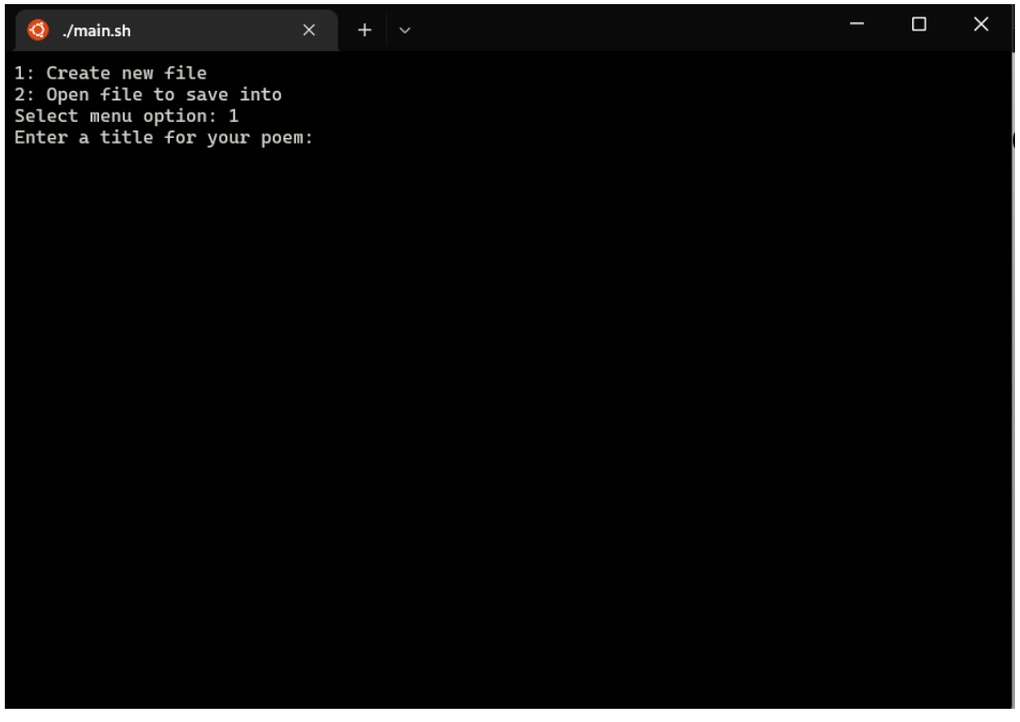
Terminal app overview

What is the Haiku Creator?

For my terminal app project, I decided to create an app that would allow the user to create haiku's and save them within a file system.
The Haiku creator starts in the main menu section, where the users can select between creating haiku's, viewing/ editing and deleting files within saved files, and has a jumbler feature that creates a new poem using the lines and titles from poems within a selected save file







Main Features and How to use the app

The main features of the terminal app, is its menu system, create a haiku feature, the file system and the jumbler feature.

Menu system: For the menu system I wanted to design something that was easy to read, easy to navigate and kept the terminal clean. The menu runs on a while loop which is essentially the main loop that keeps the app running until the user wants to quit. All inputs for the menu options are taken as integers and there is conditionals to prevent the user from entering incorrect values. The menu has a few sub menus for when accessing other parts of the app which run on the same style of system.

Create a haiku: The create a haiku feature is the main feature of the app. This was implemented to allow the user to create their own haiku's and save them within a new file or a previously saved file. To use this feature the user must go into the create a haiku menu option, from there they can select to save into a previous file or to create a new file. Once that has been selected the user is then prompted to fill out a title and the lines for the poem.
The poem creator also has a behind the scenes feature in which it checks the users input for each line and evaluates if they have met the syllable condition (which is 4-6 for lines one and three and 6-8 for line two).

File system: The file system is another crucial feature for my app, without it you would not be able to do a whole lot. The system itself is compromised of some essential functions that allow the user to save a file, open to view the poems within a file and edit certain lines of a poem, and allows the user to remove/delete a file if they choose too. The file system uses .json files to store the users poem data, my decision for this was because of the way json and python handle dictionary and list data structures.
To see the file system in use, the user can access saved files from within the menu, from there they have 3 options, 1) prints out the poems of a selected file,
2) allows the user to select a file, then select the poem they want by entering the title. Following the prompts the user can enter new lines into the poem and then type done when they are finished to save the edited poem back into the json file.
3) Remove a file, this option allows the user to delete a file by typing into the terminal the file they wish to remove.

Jumbler feature: The jumbler feature is a fun little component that I wanted to add to my terminal app. This feature allows the user to select a file that has 2 or more poems stored within it. Then it takes the poems from the file, jumbles up the titles and lines, returns to the user a new poem made up of mismatched lines and titles. the user can then choose to save the poem or jumble again.

Originally my intention for this feature was to have it jumble the poems up and return an entire file of brand new jumbled poems. However during the process of building this app, I had a thought that it would be more enjoyable for the user to just produce one poem that they can keep re-jumbling until they find a poem they were happy with.

How to use the app - a quick rundown:
So the basic flow of the app is the user will set up the python environment, following the guides located within the help section or running the bash scripts.
Then creating a virtual environment to download the necessary requirements that is located in the requirement.txt file.
Once the environment is set up and ready to go the user will be able to access the app through the main.sh bash script, or alternatively if using the bash script to set up the venv then it will start the program once venv has been activated and set up appropriately.
Within the terminal app the user can navigate through the menu system by typing in the number inputs related to where they wish to go to. The user can also type back at any input to be able to stop what they are doing and return to the main menu.
Within the create a haiku feature there are several prompts to follow that allow the user to input lines of the poem, title names, create new files or save to a previous file.
The user can also access previously saved files by navigating to the saved files menu option. Within this menu users can access saved files to view poems, edit lines within a poem and delete a file.
The jumbler feature is accessed within the main menu, to use this feature the user just needs to input the file name of the file they want to produce jumbled poems from. The user can then save these poems in the same file or jumble again.
To exit the program the user needs to input 5 at the main menu to close.

Logic and code overview

The core logic of this program is presenting the user with different options, requiring the user to input their decision, translating that input into a conditional and producing the results.

The way this is done is by implementing a while loop with a start stop variable set to true. From this implementation the menu is displayed and waits for user input. The user inputs a decision and this is then checked with the conditions. For most menu options the user is required to enter a number that relates to the menu option. for example number 2 on the main menu allows the user to enter the create a haiku feature.
There is safeguards put into place to ensure that an error is not thrown when an incorrect value of data is presented.
With number related inputs a try and except block is in place that ensures that a value error does not get thrown if the user uses something that isn't an integer.

The main loop will continue to run every time the user enters a new section, goes through the various prompts depending on which menu option they have chosen, once the user finishes with the menu option the loop resets and the user is presented with the same 5 menu options.

When the user wants to exit the program, they need to be on the main menu and type the number 5. This tells the program that the start stop variable has changed to False and ends the loop.

```
while start_stop == True:
    system("clear")
    try:
        menu.main_menu()
        menu_option = menu.menu_selection()
        system("clear")

        if menu_option == 1:
            intro_printout()
            input("Press Enter to return:")
        elif menu_option == 2:
            menu.create_old()
            menu.create_haiku_system(menu.menu_selection(), save_path)
        elif menu_option == 3:
            menu.saved_files()
            menu.saved_files_system(menu.menu_selection(), save_path)
            input("Press Enter to return:")
        elif menu_option == 4:
            print(menu.jumbler_system(save_path))
            input("Press Enter to return:")
        elif menu_option == 5:
            print("Exits program!")
            start_stop = False
        else:
            print("Sorry not a valid input, "
            "please select from the options")
            input("Press Enter to return:")
    except KeyboardInterrupt:
        input("Back to main menu: ")
```

```
# Menu options for main menu, create a haiku and saved files
def main_menu():
    print("1: Intro")
    print("2: Create Haiku")
    print("3: Saved Files")
    print("4: Jumbler")
    print("5: Quit")


def create_old():
    print("1: Create new file")
    print("2: Open file to save into")


def saved_files():
    print("1: View saved poems file")
    print("2: Edit poems from a chosen file")
    print("3: Remove a file *Caution this will delete the file*")
```

Haiku creator logic:
For the Haiku creator I had to implement a few key features/functions to ensure that the feature worked as I had intended it too. Firstly I found a package from pypi called SyllaPy - created by Micheal Holtzscher. This package did the heavy lifting for me in regards to implenting a function that could count the syllables in a given string.
The idea with the create a haiku feature was to allow the user to create a new file or save to an old file. This was done by taking user input and matching it to the desired outcome.
So if the user wanted to save the poem they were going to create to an old file, the program will open up the directory and list the file names that are there. Once the user inputs the name of the file, the program will check that the file names match and then store that information to be called upon again when saving.

The creator itself gives user prompts to get the data that it needs, to begin with it gives a prompt to give the poem a title. Then it prompts the user to input each line of the poem. Using the SyllaPy counter I was able to create a function that would take the users string and check that the amount of syllables met the condition. If it didn't meet the condition, then the user would be prompted to write it out again. However if the string does meet the line condition, then the string would stored in the value side of a dictionary object.

I opted to store my poem information as a dictionary for two main reasons. I wanted to be able to store the user input values with appropriate keys that could be seen visually. Also allowing easier access for when editing lines or jumbling the poems. The other reason is that after doing some research I had decided to store the users poems into .json files, this is because of the similar structure with dictionaries and lists.

Once the poem is done the user is prompted if they would like to save or not, choosing the yes option allows the program to save the newly created poem in its desired location and back to the main menu. If the user decides they don't want to save the poem, then they are looped back to the start of inputting the poem to try again.

```
[
    {
        "title": "Even in Kyoto",
        "line_one": "Even in kyoto",
        "line_two": "hearing the cuckoos cry",
        "line_three": "I long for kyoto"
    },
    {
        "title": "Over the Wintry",
        "line_one": "Over the wintry",
        "line_two": "Forest winds howl in rage",
        "line_three": "with no leaves to blow"
    },
    {
        "title": "Lines on a Skull",
        "line_one": "life's little, our heads",
        "line_two": "Sad, redeemed and wasting clay",
        "line_three": "this chance. Be of use"
    }
]
```

```
def haiku_creator():
    poem = {}

    # Get user input to create title of poem
    poem_title = input("Enter a title for your poem: ")
    menu.check_back_statement(poem_title)
    poem["title"] = poem_title

    # Gets user input and checks syllable count
    # appends value to poem line keys
    poem["line_one"] = five_syllable_line()
    poem["line_two"] = seven_syllable_line()
    poem["line_three"] = five_syllable_line()
    system("clear")
    print(f"Title: {terminal.bold(poem_title)}\n"
          + f"{terminal.italic(poem['line_one'])}\n"
          + f"{terminal.italic(poem['line_two'])}\n"
          + f"{terminal.italic(poem['line_three'])}")
    return poem
```

```
def five_syllable_line():
    while True:
        user_input = input("Enter line: ")
        menu.check_back_statement(user_input)
        line_count = syllable_counter(user_input)
        print(line_count)
        if line_count < 4:
            user_input = \
            f"Not enough syllables, syllable count {line_count}"
            print(user_input)
        elif line_count > 6:
            user_input = \
            f"Too many syllables, syllable count {line_count}"
            print(user_input)
        else:
            return user_input
```

File system code break down:
The file system was a really interesting area to build for my terminal app. Taught me a lot about how python interacts with files and what kind of little tricks you can implement to make sure it handles files the way you want.

A small piece of code that I wrote but find it to be one of the more vital aspects of my file system is the directory_path function. I wanted to ensure that the files would be saved in a folder called saved_files.
The function is called when the user goes to create a file or access the saved files, inside the function, a variable is defined as "./saved_files". Then using the os package I took advantage of the path.isdir() function which returns a true or false boolean depending on if the path exists. I had a conditional set up that states if the path to saved files does not exist then make the directory and return the variable of the save files path.

```
def directory_path():
    save_path = "./saved_files"
    if not os.path.isdir(save_path):
        os.mkdir(save_path)
    return save_path
```

For the user to create a file to save their poems in, I wanted to give the user the ability to name their file. For this to work I needed to implement a bit of logic that would check through the users input to make sure there was no special characters within it.
To do this, without having the user input another name if the special character was in there. I implemented a translation table that took a string of special characters I wanted to not allow, took the Unicode of each specific character. Then mapped through those uni codes and implemented them within a dictionary as keys. Then I set the value to those keys as None type.
The user then enters in a name for their file, the function takes the string and checks the string to see if there are any of the special characters in there. If there is it removes the character from the string and brings back a usable string for the file name.

```
def create_file_name():
    translation_table = dict.fromkeys(map(ord, "!@#$%^&*()+=?>< ") \
    , None)
    # While loop to prevent user from entering an empty string
    while True:
        file_name = input("Enter a name for your file: ")
        menu.check_back_statement(file_name)
        if file_name.strip() == "" :
            print("file name cannot be an empty string!")
        else:
            break

    file_name = file_name.translate(translation_table) + ".json"
    return file_name
```

Another problem I faced when implementing the file system was finding a way to allow the user to edit lines within a selected poem.
There was a bit of back and forth on what was the best way to go about this problem. I was using json files to save the new poems which I found super useful as the json object is similar structure as a python dictionary.
However to allow access to the poems I had to work through a few different paths.
When the user chooses to edit a poem, they are presented with a list of the available files within the saved file directory. The user is then prompted to write the file name, which the code will check to ensure it matches with the files within the directory.
Once the program has found a matching file it returns the poems within the file. The problem I faced here was that to return the poems and give the user an option to choose which poem they wanted to edit, I needed to be able to iterate through the poems titles.
my choice here was when loading the file using json.load(), I instructed the function to check what type of data the load is. If it was seen as dictionary, then I turned the data into a list of dictionaries.
My theory on this was that I could then iterate through the files and produce the titles for each dictionary, so the user could see what to choose.
This also allowed for appending new dictionaries onto the file, as a small issue I had found when working with just json objects and python dictionaries was that when the user was to create a new poem and save to a previous file, Instead of adding to the file, the poem would overwrite the original poem.

The actual editing of the lines to a poem, follows the same core logic for create a haiku, however with a slight difference. The function runs on a while loop that asks the user to choose a line and then input the new line. Each time it prints out the poem with the new line. Once the user is finished with editing, they are prompted to enter in done, which raises a Value Error. This breaks the while loop returning the newly formed poem.
The program then takes the new poem and matches its title with the original poem within the list of poems. Then using the dictionary method update, the program updates the new poem.
Finally the program takes the poem list with the updated poem and writes it back to the file.

```
def load_appends(file_name, haiku):
    data = json.load(open(file_name))
    if type(data) is dict:
        data = [data]
    data.append(haiku)
    with open(file_name, "w") as file:
        json.dump(data, file, indent=4)
```

```
def open_read_file(file_name):
    try:
        with open(file_name, "r") as file:
            poems = json.load(file)
            return poems
    except FileNotFoundError:
        print("Something has gone wrong with the file, \
        please try again")
```

```
def title_preview(poem_list):
    if type(poem_list) != list:
        poem_list = [poem_list]

    while True:
        i = 0
        while i < len(poem_list):
            print(poem_list[i]["title"])
            i += 1
        title_choice = input("Enter the title of the "
            "poem you wish to edit: ")
        menu.check_back_statement(title_choice)

        for poem in poem_list:
            if title_choice.lower() == poem["title"].lower():
                print(f"you selected {poem['title']}")
                return poem
            else:
                pass
        input("Sorry the title was incorrect,"
        " press enter to try again")
        os.system("clear")
```

```
def poem_editor(poem):
    try:
        while True:
            pprint(poem, sort_dicts=False)
            line_input = line_selection()
            if line_input.lower() == "line_one":
                poem["line_one"] = five_syllable_line()
            elif line_input.lower() == "line_two":
                poem["line_two"] = seven_syllable_line()
            elif line_input.lower() == "line_three":
                poem["line_three"] = five_syllable_line()
            else:
                line_input = """Please select the line you want by
                typing line_one, line_two or line_three"""
                print(line_input)
    except ValueError:
        input("Press enter to save:")
        return poem
```

```
def line_selection():
    print("""Enter the line you want to edit by typing
    line_one, line_two or line_three, type done when finished""")
    choice = input("")
    menu.check_back_statement(choice)
    if choice.lower() == "done":
        raise ValueError

    return choice
```

```
def poem_update(poem_list, updated_poem):
    if type(poem_list) is dict:
        poem_list = [poem_list]
    try:
        for poem in poem_list:
            if poem["title"] == updated_poem["title"]:
                poem.update(updated_poem)
                break
            else:
                pass
        return poem_list
    except KeyError:
        print("Something has gone wrong with the poem, " \
         "please try again")

def file_update(file_name, updated_poem_list):
    try:
        with open(file_name, "w") as file:
            json.dump(updated_poem_list, file, indent=4)
    except FileNotFoundError:
        print("Something went wrong with the file name, " \
         "please try again:")
```

Jumbler logic:
I also just want to touch on my code for the jumbler as I found it really interesting to build and how to approach it.
So originally my idea was to jumble all the poems up line by line, however when I started working on jumbler I had a slight change of mind and wanted to just jumble the whole thing up and return a new poem that mixes the titles and lines together.
So to get this to work the program asks the user to select a file from the saved files directory. Then it loads the file and checks the length of the array/list. If the array has less than 2 poems then it brings back an error message explaining that the file needs 2 or more poems to be able to jumble.
When the user selects a file with the appropriate amount of poems, the program then passes the list of poems as an argument to the jumbler function.
The jumbler function creates two empty lists one defined as dictionary keys and the other as list values. Using a for loop, the program cycles through each key value in the dictionary, putting the keys in dictionary keys variable and dictionary values into list values. Then using the random.shuffle() method on the list of values, the list jumbles the order of values.
The values are then added together with the dictionary keys using the dict(zip()) method. this allows us to create a new dictionary using the key value system I had already implemented.
The terminal prints out the new poem and asks if the user wants to save, if yes the poem appends to the list of poems and saves to the file, cycles back to the jumbler.
If no the user is prompted to jumble again.

```python
def randomiser(poem_data):
    list_values = []
    dictionary_keys = []
    for poem in poem_data:
        list_values += poem.values()
        dictionary_keys += poem.keys()
    shuffle(list_values)
    shuffled = dict(zip(dictionary_keys, list_values))
    pprint(shuffled, sort_dicts=False)
    return shuffled
```

```
['trial_two.json', 'sample_poems.json']
Enter the name of a file you want: sample_poems.json
{'title': 'Over the wintry',
 'line_one': 'Forest winds howl in rage',
 'line_two': 'hearing the cuckoos cry',
 'line_three': "life's little, our heads"}
Would you like to save this poem?
 y or n: n
No worries, jumble again!
{'title': 'Lines on a Skull',
 'line_one': 'Over the Wintry',
 'line_two': "life's little, our heads",
 'line_three': 'Sad, redeemed and wasting clay'}
Would you like to save this poem?
 y or n: n
No worries, jumble again!
{'title': "life's little, our heads",
 'line_one': 'I long for kyoto',
 'line_two': 'Over the wintry',
 'line_three': 'hearing the cuckoos cry'}
Would you like to save this poem?
 y or n:
```

Development review

The development process of this terminal app went in a few different stages, which I found really interesting and was able to really dive into different areas of problem solving.
Initially coming up with a haiku creator was something that I had a lot of passion in, I'm a really big fan of poetry and also enjoy making my own poems. This formed the idea that it would be fun to build a terminal app that allows a user to create their own poems and to edit,view and jumble as they like.
From the idea I needed to start making a solid project management plan, This is actually something I have not really touched on before, so it was really good to learn how to plan each step out, keep track of what was happening and how I was tracking.
I found that with some parts of my project management I ended up having to extend deadlines just due to a certain challenge that I may of faced.

Some of the more interesting and difficult challenges I faced while building this application were: Implementing the logic and code for being able to access saved files, edit them and then save the file again. When first doing so I couldn't really understand how to save a new dictionary within a previous file or to iterate over the dictionary keys to get the titles to be displayed on the terminal. After a bit of research I had figured that it would be more beneficial to convert the dictionaries into a list of dictionaries, allowing for iteration and to append new saved objects.
Another interesting challenge was error handling and testing. For the error handling, I hopefully covered all the potential bugs/errors but it was really fun just learning how to raise exceptions to break certain loops or to catch exceptions with try and except blocks to prevent the whole program from throwing and error and breaking. Pytest was a really interesting aspect as well, testing took a bit of understanding and taught me a lot about trying to ensure functions only do one thing. Something I learnt a bit about that I found really good was how to monkeypatch to set attributes for user input. It made automating tests and producing the results I was looking for really enjoyable.

Looking back on this project I have learnt a lot and really look forward to diving in more with programming. Some aspects that I would like to hone in on is object orientated programming as I believe it would of helped me a lot with keeping my code dry and clean. Although I was a bit hesitant just due to not fully understanding OOP just at this moment.