HW 1: Computer Convo

CS550, Eric Yang

Skills Practiced: NumPy, Math, Random, Art, Time, TextBlob(SentiAnalysis), OOP

Designing the Greeter

This project doesn't necessarily ask for an object-oriented design; however, I wanted to have an **infinite interaction loop** within the program so an object-oriented design with reusable methods is the way to go. For example, we can have a ***tell_joke()*** method in a ***Greeter*** class that can be reused to tell random jokes. To initialize the program, lets build a skeleton structure for a basic ***Greeter*** object.

```
class Greeter:
    def __init__(self, user_name):
        self.name = user_name
        self.suffixes = ["..."]
        self.random_questions = ["..."]
        self.random_jokes = ["..."]
        self.random_jokes_ans = ["..."]


    def interaction(self):
        pass

    def play_ericsweeper(self):
        pass

    def analyze_emotion(self, response):
        pass

    def make_nickname(self):
        pass
```

```
    def give_joke(self):
        pass

    def give_rand_question(self):
        pass

    def y_n_handler(self):
       pass

    def tell_horrible_story(self):
        pass


if __name__ == "__main__":
    pass
```

Here, we've built a basic ***Greeter*** class that has a few core methods. To initialize a ***Greeter*** we only need the user's first name. Because this is just a homework assignment, I've decided to just store the data for jokes and questions in lists. A **.pkl** file would be more suitable if we want to turn this into a project. Let's take a look at what our methods each should do:

interaction()

Arguments: none
Return: none

This method prompts the user to go through **1** interaction loop, in the "__main__", we will place this in a while True: loop to run an infinite interaction loop.

play_ericsweeper()

Arguments: none
Return: none

This is the method where we'll implement a dumbed-down version of minesweeper for the user to play

analyze_emotion()

Arguments: (String) user_input
Return: none

This method powers the sentiment analysis of the ***give_rand_question()*** method. It will take in the user input String and use a basic TextBlob model to analyze the polarity score of each text (basically how sad/happy each one is, - is bad, ~0 is neutral, + is good).

make_nickname()

Arguments: none
Return: (String) nickname

This method will make & return a nickname when called

give_joke()

Arguments: none
Return: none

This method will give a joke and reveal the answer once the user presses enter.

y_n_handler()

Arguments: none
Return: boolean

This is the handler method we will use for most yes/no inputs. **Because we need to handle yes/no input nearly infinite times during execution, a handler will be better.**

tell_horrible_story()

Arguments: none
Return: none

This will prompt the user through a sequence to enter some names, verbs (past tense), and adjectives. Then it will print a story with these things added in.

Building make_nickname()

Because of the time constraints this project has the nickname method is going to be really simple. My implementation of the method basically:

1. Takes the first **2** characters of your name
2. Draws a random suffix from the object's ***suffixes[]*** list with **random.choice()**(I really want to call this an array XD)
3. Prints a formatted String with the {front_two_head}{rand_suffix}
4. Use **art**'s **tprint** to print a more artsy version of it

```
def make_nickname(self):
    if len(self.name) ⩽ 3:
        return "Sorry! Your name is too short"
    else:
        front_two_head = self.name[:2].lower()
        rand_suffix = random.choice(self.suffixes)
        nickname = f"{front_two_head}{rand_suffix}"
        return nickname.capitalize()
```

Building give_joke()

This implementation is actually very similar to the nickname random suffix implementation. However, this time, we want to store the answers to our jokes in a separate answer list. This means we have to **keep track of our random index,** which means it might be better to use **random.randint(0, len(list)-1)** as a random index picker instead.

```
def give_joke(self):
    # we need an index number to get the joke ans so we cant just use
random.choice()
    random_joke_index = random.randint(0, len(self.random_jokes)-1)
    print(self.random_jokes[random_joke_index])
    time.sleep(0.8)
    print("Think you got the answer? Press enter to see!")
    input()
    print(self.random_jokes_ans[random_joke_index])
    time.sleep(0.6)
```

Building give_rand_question() & analyze_emotion()

This method also as the same idea as the nickname suffix. We store our random emotion questions in a list and then pick a random one. The only new thing here is that **because it is a question, we want to read in user input and respond to it**. Originally, I just planned to always print “thanks for telling me”, but I thought some quick sentiment analysis would make this function a lot more fun and enjoyable because that is what the rubric said after all.

```
def give_rand_question(self):
print(random.choice(self.random_questions))
print(self.analyze_emotion(input(" >>>")))
```

I am not going to get into too much detail about the sentiment analysis because most of the heavy lifting is done by the TextBlob language processing library in Python. The method is actually really simple. We initialize a TextBlob object with our user_input and then retrieve the polarity score from it. From there, we can simply use a conditional to determine which response we should get based on emotion. The current number thresholds could be tweaked further; I only did a little bit of testing. (It’s a shame Python does not have a switch built-in)

```
def analyze_emotion(self, response):
    blob_analyzer = TextBlob(response)
    polarity = blob_analyzer.sentiment.polarity
    # USE PRINT FOR TUNING ONLY
    #print(polarity)
    # this can be fine-tuned further to be more accurate with the
sentiment analysis
    if polarity > 0.5:
        return "That's amazing to hear!"
    elif polarity > 0.2:
        return "That's great to hear!"
    elif polarity > 0:
        return "One of those regular days, huh?"
    elif polarity > -0.5:
        return "I hope you feel better soon!"
    else:
        return "No matter what is going on right now, know that
everything will be okay!"
```

Now that we have “random” imported, we can use its randInt(start, end) function to generate an integer between the 0th index of the list and the length of the list - 1(AKA the last index of the list)

Building y_n_handler() & tell_horrible_story()

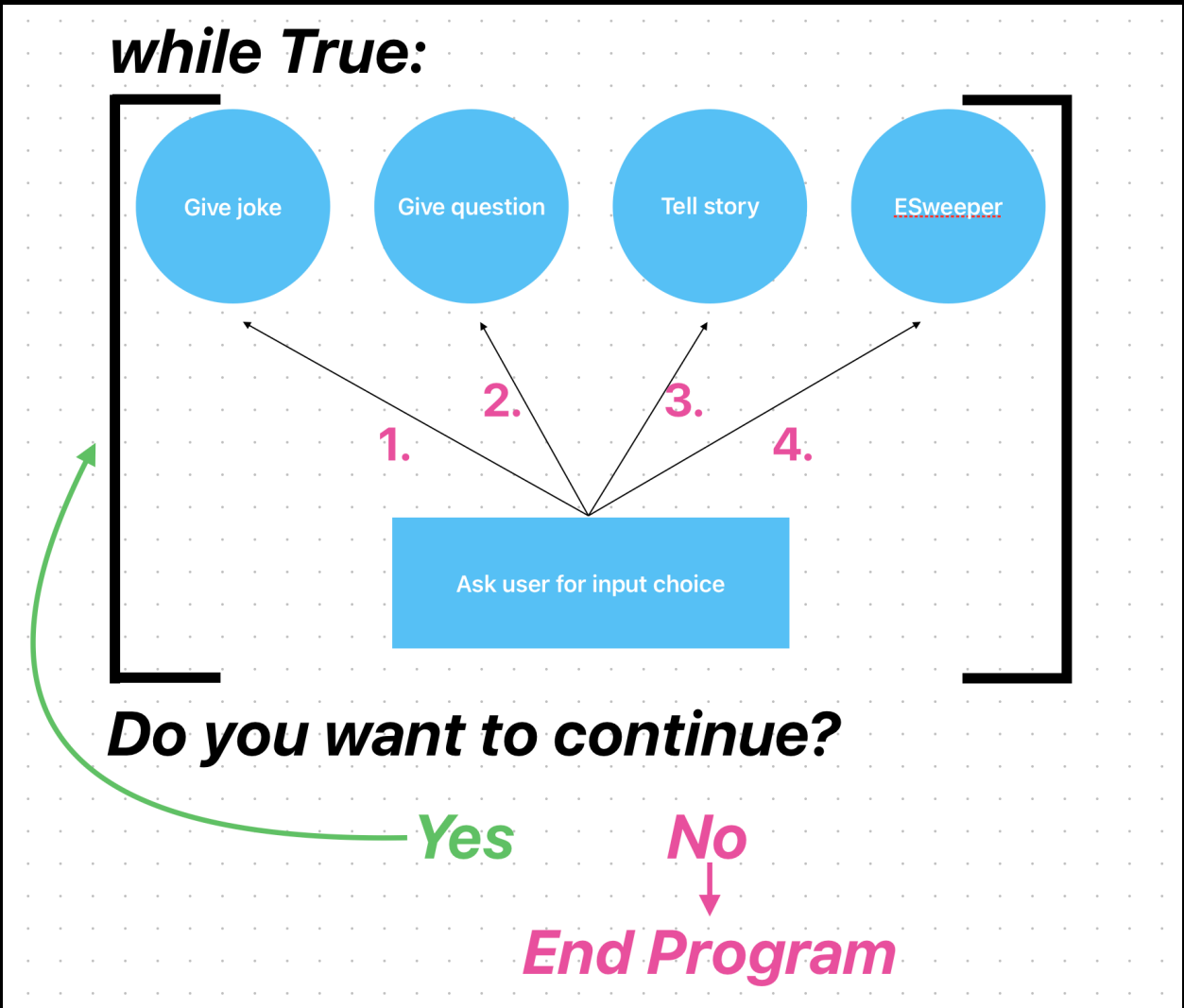
The ***y_n_handler()*** method is just a handler method you will use after you print a question somewhere in the code. It basically prompts the user with a y/n input and then returns **True** if the user said yes and **False** otherwise.

```
def y_n_handler(self):
    choice = input("(y/n)\n >>>")
    choice.lower()
    if choice == "y":
        return True
    else:
        return False
```

The ***tell_horrible_story()*** method uses multiple calls of **input()** to ask the user to input a set of names, verbs, and adjectives to make a story. It prints out a formatted string of the fill-in-the-blank story but is now filled with whatever the user entered. **A cool extension to this could be integrating the OpenAI API into your code and prompting GPT to generate a fresh story every time with the inputs.**

Implementation will not be shown for this method as it is very String text heavy & basic. Check canvas or GitHub for this method's code

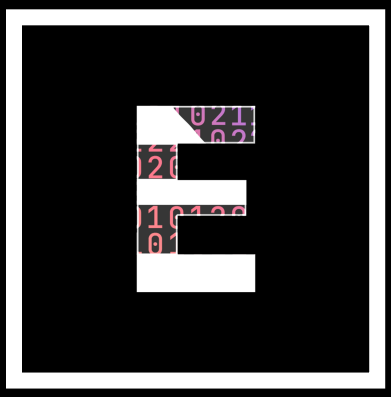
Building interaction()



The graph above explains the flow of the program, inside the **while True:** loop, we call ***interaction()*** every, single, time. The interaction method asks the user to input a number for the action they want to take this time around. Then, it will call its own methods accordingly. In the main method, we just have to use a **Greeter** object in a while True: loop to complete this interaction.

```
def interaction(self):
    interaction_choice = input("1. Get a joke, 2. Get a random
question, 3. Get a really, really horrible story XD, 4. Play
Ericsweeper\n >>>")
    if interaction_choice == '1':
        self.give_joke()
    elif interaction_choice == '2':
        self.give_rand_question()
    elif interaction_choice == '3':
        self.tell_horrible_story()
    elif interaction_choice == '4':
        self.play_ericsweeper()
    else:
        print("Oops! That's not an option!")
```

```
if __name__ == "__main__":
    first_name = input("Hello! What is your first name?\n >>>")
    main_greeter = Greeter(first_name)
     *additional pre-setup implementation not shown here*
       while True:
        main_greeter.interaction()
        keep_going = input("Do you want to keep talking? (y/n)
\n >>>")
        if keep_going == "n":
            print("Bye! Thanks for using my program!")
            break
```



About EricSweeper

A quick game that I made to go along with this project. It uses NumPy's 2D array to literally simulate a grid. The full code can be found on the GitHub. I initially programmed this as a separate file but put it in as part of the ***Greeter*** class.

On my honour, I have neither given nor received unauthorized aid - Eric Yang

External sources consulted:

I learned about and used the ASCII art module from to present prettier nicknames: https://pypi.org/project/art/

I used Textblob for basic sentimental analysis to aid in the creation of the infinite interaction loop: https://textblob.readthedocs.io/en/dev/quickstart.html#sentiment-analysis

Polarity score reading I did to learn more about how to set my polarity thresholds for positive and negative emotions: https://www.red-gate.com/simple-talk/development/data-science-development/sentiment-analysis-python/

I used ChatGPT to generate the story cutout templates with empty names and adjectives

I used numpy for the Ericsweeper thing: https://numpy.org/doc/stable/reference/arrays.ndarray.html