Bash Script

Here's a brief overview of what each script does:

1. python3.sh: Checks if Python 3 is installed. If not, it opens the Python download page in your default browser.
2. env.sh: Creates a Python virtual environment in the src directory.
3. requirement.sh: Installs or updates the specified Python packages in the virtual environment src/myenv.
4. bash.sh: Changes to the src directory, ensure that python3.sh,env.sh and requirement.sh all have execute permissions first,checks if Python 3 is installed, creates a virtual environment if Python 3 is installed, activates the virtual environment, installs or updates the specified Python packages, runs main.py with Python 3, and deactivates the virtual environment.

User will be advised to first ensure that they are on a Unix-like systems (such as Linux and macOS) terminal and type the following to ensure that they first have execute permission.

chmod a+x bash.sh

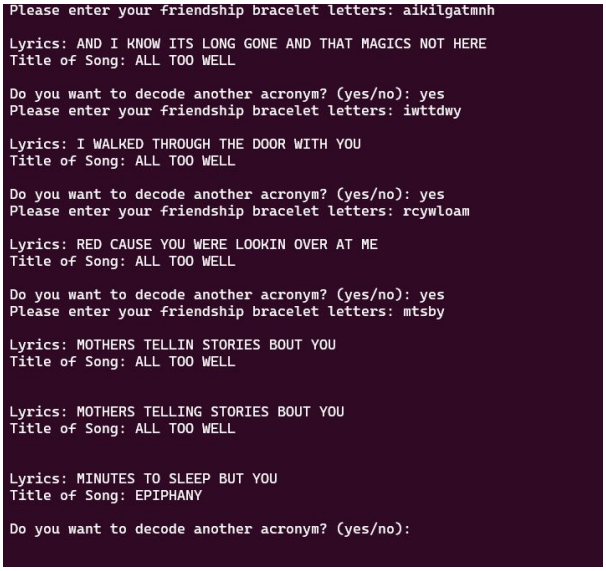
Walk-Through of application

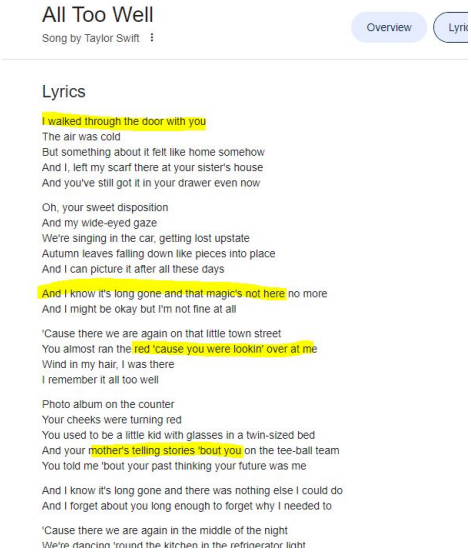
```
alicet@DESKTOP-8NHATH1:~/projects/projectb$ ./bash.sh
Python 3 is installed.
Virtual environment created in directory myenv.
Installing/updating beautifulsoup4==4.12.2
Collecting beautifulsoup4==4.12.2
  Using cached beautifulsoup4-4.12.2-py3-none-any.whl (142 kB)
Collecting soupsieve>1.2
  Using cached soupsieve-2.5-py3-none-any.whl (36 kB)
Installing collected packages: soupsieve, beautifulsoup4
Successfully installed beautifulsoup4-4.12.2 soupsieve-2.5
Installing/updating certifi==2023.7.22
Collecting certifi==2023.7.22
  Using cached certifi-2023.7.22-py3-none-any.whl (158 kB)
Installing collected packages: certifi
Successfully installed certifi-2023.7.22
Installing/updating charset-normalizer==3.3.0
```

```
All packages have been installed or updated successfully.
Please enter your friendship bracelet letters:
```

Once user run the ./bash.sh command on terminal, the script will check and download all requirement to run application before starting. The first user prompt output will be the “Please enter your friendship bracelet letters:”.

Walk-Through of application





This application is designed to decipher acronyms found in song titles or lyrics. It can extract lyrics from any part of a song,no matter the length. For instance, a line from the lyrics can be fetched from any section within the complete lyrics. If an acronym corresponds to a lyric from a song, the application will also display the title of the song from which the lyrics were derived.

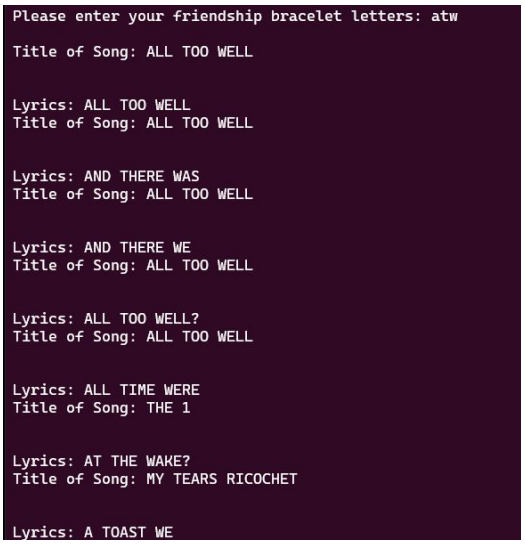
Walk-Through of application

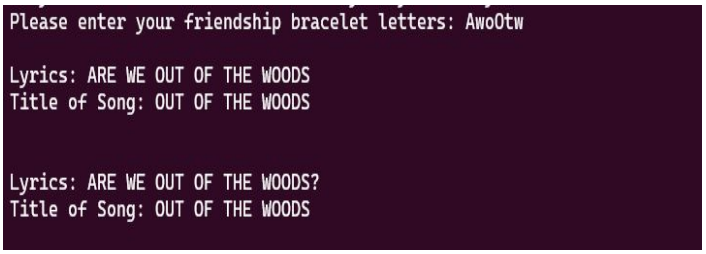
```
Please enter your friendship bracelet letters: inthaf

Title of Song: ITS NICE TO HAVE A FRIEND


Lyrics: ITS NICE TO HAVE A FRIEND
Title of Song: ITS NICE TO HAVE A FRIEND
```

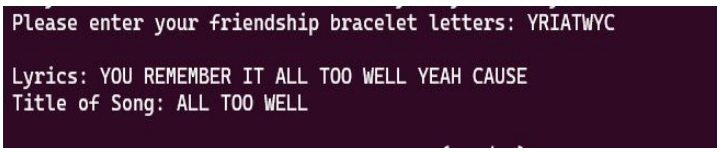
```
Please enter your friendship bracelet letters: tosotd

Title of Song: THE OTHER SIDE OF THE DOOR


Lyrics: THE OTHER SIDE OF THE DOOR
Title of Song: THE OTHER SIDE OF THE DOOR
```

The application is also capable of identifying matching acronyms in song titles, provided the title is more than two words long. This is due to the fact that shorter song titles are likely to be spelled out entirely on a friendship bracelet, eliminating the need for decoding. You may notice the application displays both the song title and the corresponding lyric and the corresponding lyric song title on separate line. This is because it identifies matching song titles and if the title also appears in the lyrics, it will be returned as a matching acronym as well.

Walk-Through of application

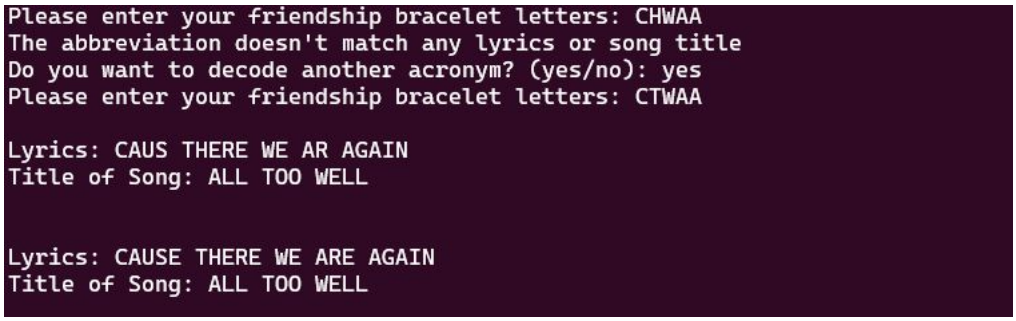


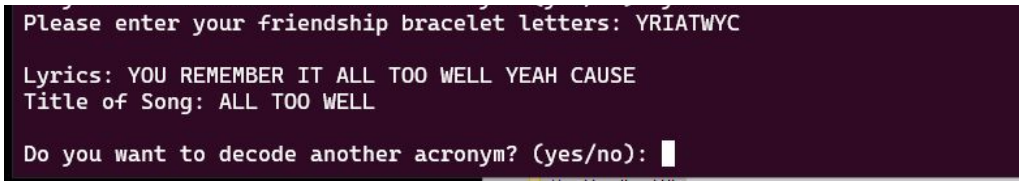




User input can be in capital or lowercases and terminal will print out corresponding full song titles or lyrics or both, according to the acronyms inputted.

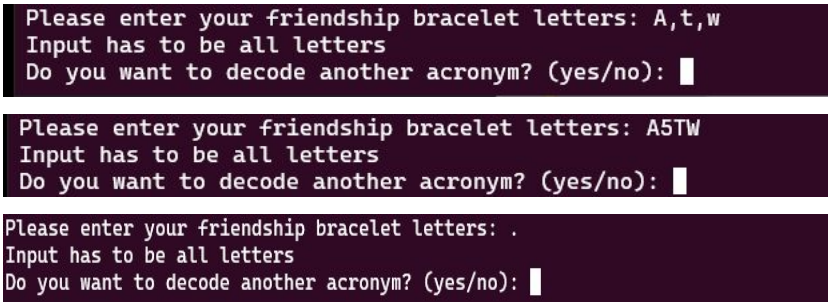
Walk-Through of application





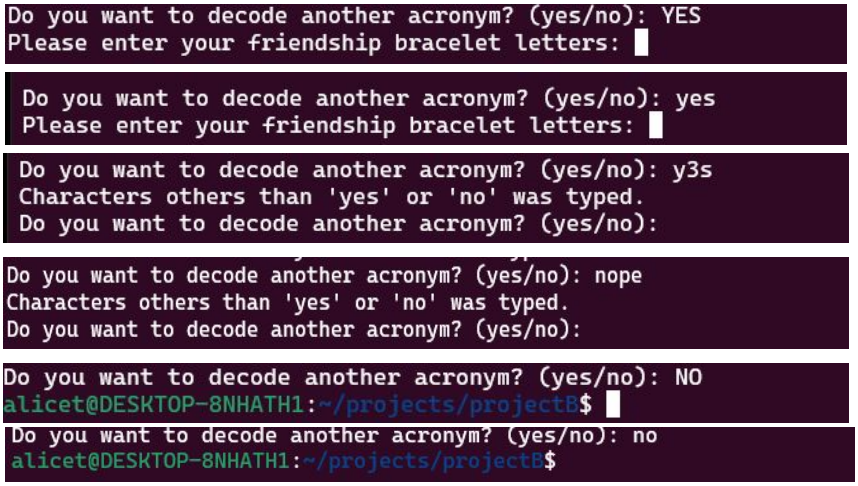
Should there be no matches to the acronym in my database that it can connect to a full set of lyrics or song titles then an output message will say so and reprompt user to ask if they still want to decode another bracelet. Should there be a match it would still ask you if you want to decode another bracelet.

Walk-Through of application



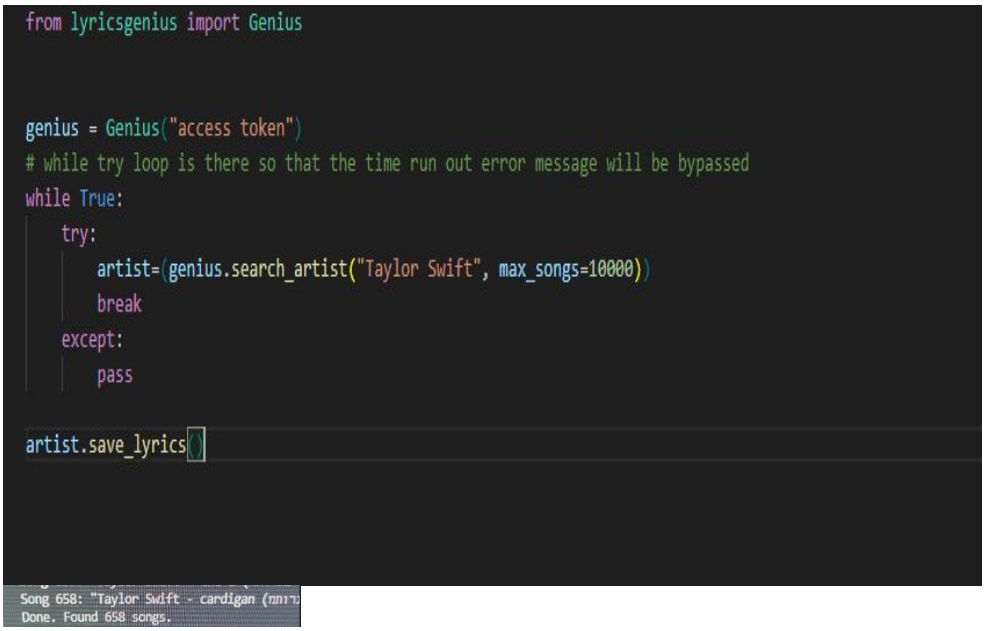
In terms of the prompt “Please enter your friendship bracelet letters:”, if anything else but a letter is inputted an error message will print out and it will reprompt again.

Walk-Through of application



In terms of the prompt “Do you want to decode another acronym? (yes/no):”, user can input ‘yes’ or ‘no’ in Capital and it will still understand the input. However if anything else then ‘yes’ or ‘no’ in capital or lowercase is inputted,application will not terminate but an error message will print out and prompt the question again. Only when ‘no’ or ‘NO’ is typed will the application terminate.

Function : Scrapping songs catalogue data



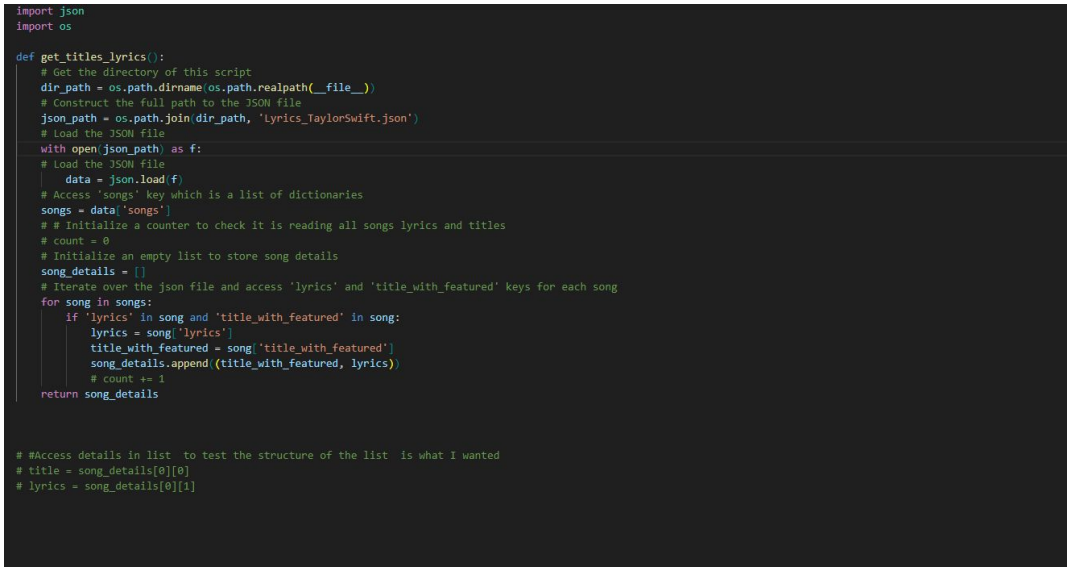
The first feature I had to code was a function to scrape song data and save it into a file for reading. This was accomplished using an existing Python library, which saved me from having to code a web scraping function from scratch. However, I quickly realized that following the standard documentation for the library was insufficient due to the large number of songs Taylor Swift has.

During the scraping process, a timeout error message was raised after a certain number of songs were scraped. To handle this error, I used a try-except block to bypass it. If an error occurred, the code would execute the “pass” statement, which would ignore the error message and retry fetching the data until it succeeded.

I set max_song = 10000 as an arbitrary number that I knew would capture all her songs. Since she has fewer than 10,000 songs, this would return her entire catalogue. In the end, data for 658 songs were retrieved from the Genius API.

The data was then saved into a JSON file using the .save_lyrics() function from the LyricsGenius library.

Function : Reading Json file



```python
import json
import os

def get_titles_lyrics():
    # Get the directory of this script
    dir_path = os.path.dirname(os.path.realpath(__file__))
    # Construct the full path to the JSON file
    json_path = os.path.join(dir_path, 'Lyrics_TaylorSwift.json')
    # Load the JSON file
    with open(json_path) as f:
    # Load the JSON file
        data = json.load(f)
    # Access 'songs' key which is a list of dictionaries
    songs = data['songs']
    # # Initialize a counter to check it is reading all songs lyrics and titles
    # count = 0
    # Initialize an empty list to store song details
    song_details = []
    # Iterate over the json file and access 'lyrics' and 'title_with_featured' keys for each song
    for song in songs:
        if 'lyrics' in song and 'title_with_featured' in song:
            lyrics = song['lyrics']
            title_with_featured = song['title_with_featured']
            song_details.append((title_with_featured, lyrics))
            # count += 1
    return song_details


# #Access details in list  to test the structure of the list  is what I wanted
# title = song_details[0][0]
# lyrics = song_details[0][1]
```

Since my LyricsGenius library code saved the songs data into a JSON file, I used the Python standard library json to load and read from the file. Once I found which key contained the lyrics and titles, I used a for loop to access all songs.

I decided to structure my data as lists within a list called songs_details. The separation of each song with its corresponding title was crucial. I needed a systematic method to pair the lyrics with the song title for my decoder terminal output.

This structure also allows me to loop through each song and manipulate the data for future functions. I also initialized a count check to make sure that the for loop collected all 658 songs data.

I checked the structure of my first song list to ensure that I understood how the title and lyrics could be accessed using their appropriate indices. This is crucial for ensuring that my decoder functions correctly in subsequent steps.

Finally, in doing my bash script I noticed that if I wanted to run my main code through the batch file and terminal I would have to use a different path for the Json file to be read but in changing so I couldn't run in Vscode. When you run the script from VS Code, it sets the working directory to the location of the file you're running. However, when you run your bash script from the terminal, it changes to the src directory before running main.py. Therefore I imported os module to dynamically get the directory of your current Python script and construct the path to Lyrics_TaylorSwift.json. That way it would work both through the bash script and vscode.

Function : User input

```python
def user_input():
    try:
        letters_to_decode = input("Please enter your friendship bracelet letters: ")
        # Converting each letter from user input to a list
        letters_to_decode_list = list(letters_to_decode)
        # Make sure that the input will be converted to upper case so that it will match title_details format
        letters_to_decode_list = [letter.upper() for letter in letters_to_decode_list]
        # Check if input list element are all letters if not true then an exception error will be raised
        if not all(item.isalpha() for item in letters_to_decode_list):
            raise ValueError("Input has to be all letters")
    except ValueError as e:
        print(e)
        return None
    return letters_to_decode_list
```

To decode the user's input, I first prompt the user to provide their input. This input is then converted into a list, which allows me to iterate over each letter. This is crucial for my decoding functions, as it enables me to manipulate each letter individually and compare values using indices.

By converting the input into an iterable list, I did my first mutation by transforming each letter into uppercase using a 'for in' loop. This ensures consistent formatting and prevents case sensitivity issues. For instance, if my Json data file contains uppercase letters and the user inputs lowercase letters, this could lead to mismatches during decoding.

To further ensure the accuracy of the decoding process, I implemented an error handling mechanism using a try-except block. This block checks the user's input and raises a ValueError if it contains any non-letter characters. The user is then informed that the decoder only accepts letters and not numbers or special characters.

If no errors are detected, the function returns the user's input as a list of uppercase letters.

Function : Text formatting

```python
import re

def text_formatting(text):
    text = re.sub(r"\(.*?\)|\[.*?\]|/|,|\"|'|(\.\.\.)|\.", "", text)
    text = re.sub(r"-"," ",text)
    # Remove unicode from utf-8 files by converting to ascii and then back to its original form
    text = text.encode("ascii", "ignore").decode()
    text = text.upper()
    words = text.split()
    return text,words
```

After generating my song lyrics and titles, I noticed that some basic text formatting cleanup was necessary for my input decoder, which only accepts letters, to function properly. By creating a separate function for this cleanup, I was able to avoid repeating the same process for both my lyrics and titles, thus adhering to the DRY principle.

To assist with this task, I utilized the re module from Python's standard library twice. This module allowed me to check if a specific string matched a given regular expression. For the first instance I replace it with an empty string, effectively erasing it. It's important to note that my initial attempt to remove anything that wasn't a letter, a space, or a hyphen resulted in some issues with the lyrics output. As a result, I had to refine my approach.

Here's what I removed and why:

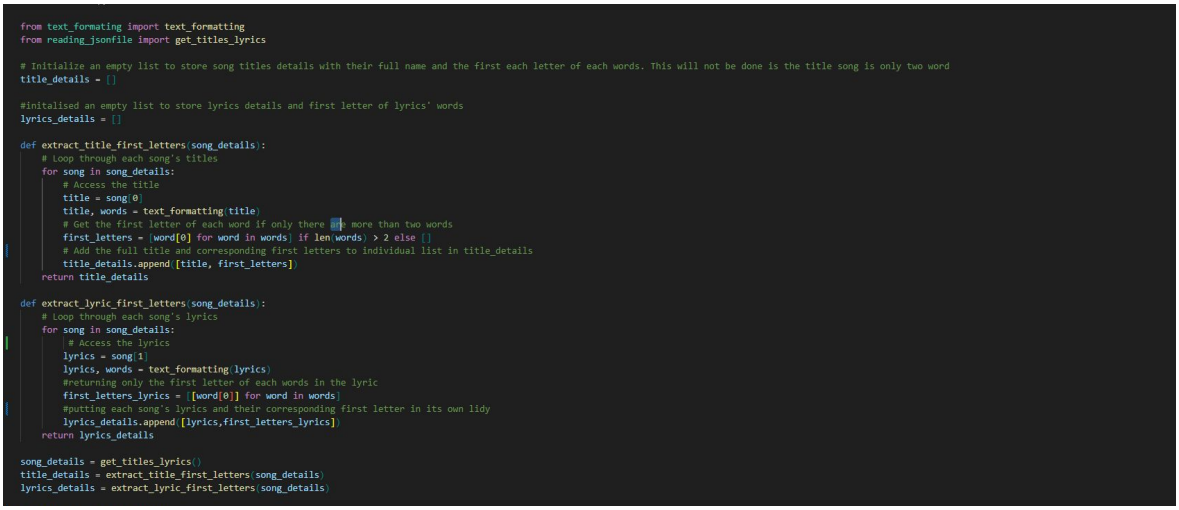
- Anything enclosed in parentheses (.*?) or square brackets \[.*?\], as well as forward slashes /. Some titles included terms like (remix), [remix], or slashes, which I didn't want to include. By removing these elements, I could ensure that titles like "All too well (remix)" and "All too well" would both be processed as "All too well" for future function, eliminating duplicates in my terminal output . Additionally, I didn't want to include elements like "(Oh-oh-oh)" in the lyrics as they aren't actual lyrics acronyms to decode.
- Commas ,, double quotes \", and single quotes '. These characters shouldn't be counted as indices in my list since the input decoder only accepts letters. For instance, in the lyric: "I say, 'I hate you,' we break up, you call me, 'I love you'", these characters would be removed.
- Ellipsis \.\.\.. During testing of one song, I realized that I needed to remove triple dots to correctly process titles like: "…Ready for it?".

My second re.sub() was to check for all instance of hyphens but this time replace it with a space and not an empty string because of songs title and lyrics such as "Anti-Hero". When splitting each words,the lack of space meant that it only capture the letter "A", however when decoding the lyrics I wanted "AH" to return "Anti Hero" match.

I then had to deal with erasing all unicode in the data to further guarantee only lyrics letters would be captured.

Once I was satisfied with the cleanup and had conducted several tests, I split the cleaned-up text. This allowed each word to become an element of a list which could then be iterated over and manipulated. This step would enable me to extract the first letter of each word for my future decoding function.

Function : Extracting the first letters



```
from text_formating import text_formatting
from reading_jsonfile import get_titles_lyrics

# Initialize an empty list to store song titles details with their full name and the first each letter of each words. This will not be done is the title song is only two word
title_details = []

#initalised an empty list to store lyrics details and first letter of lyrics' words
lyrics_details = []

def extract_title_first_letters(song_details):
    # Loop through each song's titles
    for song in song_details:
        # Access the title
        title = song[0]
        title, words = text_formatting(title)
        # Get the first letter of each word if only there are more than two words
        first_letters = [word[0] for word in words] if len(words) > 2 else []
        # Add the full title and corresponding first letters to individual list in title_details
        title_details.append([title, first_letters])
    return title_details

def extract_lyric_first_letters(song_details):
    # Loop through each song's lyrics
    for song in song_details:
        # Access the lyrics
        lyrics = song[1]
        lyrics, words = text_formatting(lyrics)
        #returning only the first letter of each words in the lyric
        first_letters_lyrics = [[word[0]] for word in words]
        #putting each song's lyrics and their corresponding first letter in its own lidy
        lyrics_details.append([lyrics,first_letters_lyrics])
    return lyrics_details

song_details = get_titles_lyrics()
title_details = extract_title_first_letters(song_details)
lyrics_details = extract_lyric_first_letters(song_details)
```

The code initiates with the import of two functions: 'text_formatting' from the module of same name, and 'get_titles_lyrics' from the 'reading_jsonfile' module. To begin looping through my lyrics and titles dataset, I must first access the list 'song_details' from function 'get_titles_lyrics'. Furthermore, to guarantee correct formatting of my data prior to extracting initial letters from words accessing the 'text_formatting' function is also necessary.

Following this, it establishes initialization for song title and lyrics details storage by creating two empty lists, 'title_details' then 'lyrics_details' respectively.

The function 'extract_title_first_letters' iterates through each song and if this title comprises more than two words, it then extracts the first letter from every word. Subsequently appending these details to the list 'title_details'. This if statement was added as there is no necessity to abbreviate two-word titles on a bracelet; hence, I dismissed the need for decoding such length titles.

I define a similar function as 'extract_lyric_first_letters'; it processes song lyrics in an identical manner. The key distinction, however, lies in its operation: irrespective of the length of the lyrics , it loops through each word and stores details within the 'lyrics_details' list.

I format the 'title_detail' and 'lyrics_details' output as nested list structure. Storing data in this format facilitates future decoding. For instance, if a sequence of first letters was found and I seek to identify the corresponding complete titles or lyric for my output. I would merely iterate over either 'title_details' or 'lyrics_details matching those initial characters. Upon finding a match, I can effortlessly access the full title or lyrics as they reside together in an sublist.

The code concludes with a call to the 'get_titles_lyrics' function, retrieving song details. It then invokes

the two pre-defined functions for processing and storing title and lyric details.

The guiding principle in software development, known as "Don't Repeat Yourself" (DRY), dictates the avoidance of duplicating code. Despite this, maintaining two separate functions for this extracting of two letters I thought could be justified due to valid reasons. Each function rigorously adheres to the Single Responsibility Principle (SRP) - they have distinct tasks: ‘extract_title_first_letters’ operates on song titles while ‘extract_lyric_first_letters’ manipulates song lyrics. Such segregation significantly improves both readability and maintainability of the codebase. It provides flexibility for future modifications: should the processing of titles and lyrics need to diverge, separate functions offer easy adjustments without impacting the other function. For instance, in my titles function if a title has more than two letters,I only want it to capture the first letter; however with lyrics, I aim at capturing all words' initial letters. Finally, isolating functions can streamline debugging; this allows issues to be traced and resolved in a more targeted manner. I considered these factors while balancing DRY principle to help with maintainability.

Function : Ordered subset for lyrics

```python
def locate_ordered_lyric_subsets(small, large):
    matches_in_song = []  # List to store all matches
    # turns first_letters_lyrics into one list with each letters as element of one list
    large = [item for sublist in large for item in sublist]
    len_small = len(small)
    # loop that iterates over the total length of the large list by the length of the small list at each iteration. The "+1" is there because Python's range() function stops one number before the end.
    for i in range(len(large) - len_small + 1):
        # checks if the current slice of the large list is equal to the small list to confirm that small is an ordered subset of large. This is done by comparing the value of the indices.
        if large[i:i+len_small] == small:
            # if true immedially returns the starting index and the ending index of the matching slice by calculating the i+the lengh of the small index to capture the end indices.
            matches_in_song.append([i, i+len_small])  # Add the match to the list
    return matches_in_song if matches_in_song else False  # Return all matches if any, else False
```

In the process of decoding lyrics, I identified the need for a function that could verify if a smaller list is a subset of a larger list. This became particularly important when dealing with partial lyrics input. While the length of a song title first letter input would exactly match the length of the song title first letter, lyrics input could be just a portion of the full lyrics. Therefore, I needed a mechanism to confirm if this partial input is indeed part of the complete lyrics. Moreover, it was crucial that this match wasn't just based on the presence of letters, but also their order. The function was designed to ensure that the sequence of letters in the letters_to_decode_list input exactly corresponds to a sequence in the first_letters_lyrics.

'Locate_ordered_lyric_subsets(small, large) 'systematically finds all instances of an ordered subset (small) within a more extensive list (large): it commences with the creation of an empty list to store these matches called "matches_in_song." The next step involves having to flatten the larger list which would be 'first_letters_lyrics' into one single list so that each letters are an element of one list.This way it makes checking matches by indices less complicated, since my input list 'letters_to_decode_list' is a flat list but my original 'first_letters_lyrics' list is a sublists.

The length of the small list (my input list 'letters_to_decode_list') is then stored in len_small. The function then iterates over the large list, checking each slice of length len_small to see if it matches the small list.

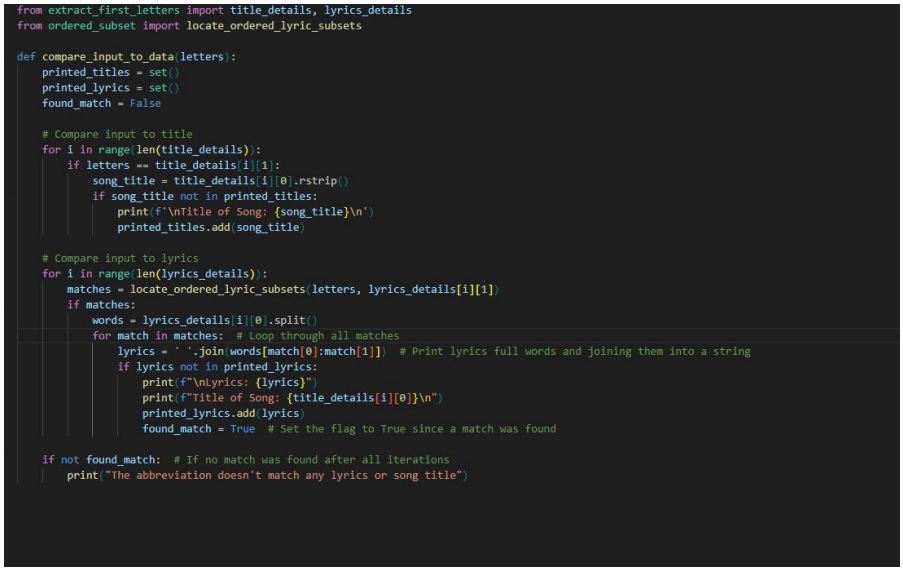
I purposefully made it loop through the large subset for matches so that it wouldn't find only the first match in a song in case input acronyms matched different lyrics. I wanted to captured them all before I start to remove duplication in my future function.

The line if large[i:i+len_small] == small: is where the magic happens. This line is inside a loop that iterates over the large list. For each iteration, it takes a slice of the large list starting at index i and ending at index i+len_small. This slice is a sublist of the large list that has the same length as the small list.

The == operator then compares this slice with the small list. If every element in the slice matches the corresponding element in the small list (i.e., the first element of the slice matches the first element of small, the second element matches the second, and so on), then the == operator returns True. This means that an exact match for small has been found within the large list, starting at index i.

If a match is found, then the start and end indices of the matching slice (i and i+len_small) are appended to the matches_in_song list. These indices represent the location of the match within the original large list.

Function : Compare input to data



```python
from extract_first_letters import title_details, lyrics_details
from ordered_subset import locate_ordered_lyric_subsets

def compare_input_to_data(letters):
    printed_titles = set()
    printed_lyrics = set()
    found_match = False

    # Compare input to title
    for i in range(len(title_details)):
        if letters == title_details[i][1]:
            song_title = title_details[i][0].rstrip()
            if song_title not in printed_titles:
                print(f'\nTitle of Song: {song_title}\n')
                printed_titles.add(song_title)

    # Compare input to lyrics
    for i in range(len(lyrics_details)):
        matches = locate_ordered_lyric_subsets(letters, lyrics_details[i][1])
        if matches:
            words = lyrics_details[i][0].split()
            for match in matches:  # Loop through all matches
                lyrics = ' '.join(words[match[0]:match[1]])  # Print lyrics full words and joining them into a string
                if lyrics not in printed_lyrics:
                    print(f"\nLyrics: {lyrics}")
                    print(f"Title of Song: {title_details[i][0]}\n")
                    printed_lyrics.add(lyrics)
                    found_match = True  # Set the flag to True since a match was found

    if not found_match:  # If no match was found after all iterations
        print("The abbreviation doesn't match any lyrics or song title")
```

This Python script is designed to compare a given input string of letters to song titles and lyrics. It starts by importing title_details and lyrics_details from the extract_first_letters module, and the function locate_ordered_lyric_subsets from the ordered_subset module.

It initializes three variables; printed_titles and printed_lyrics as empty sets and found_match is set to False. Later, we will utilize this to confirm if a match was found.

The script iterates over the length of title_details: it compares the input string to the second element of each tuple in title_details (first_letters_titles). If a match is found, trailing spaces from this specific tuple's first element(full song title) are stripped then an examination follows, has this song been printed previously? If not yet recorded as such, printing occurs and addition to set happens simultaneously. I added the rstrip as I noticed there was some duplicates due to spaces in the titles.

Next, it iterates over the length of 'lyrics_details'; for each lyric_detail the function locate_ordered_lyric_subsets is called with two inputs: the input string and second element from within 'lyrics_details' tuple ('first_letters_lyrics'). Upon finding matches , a word-split operation commences on the first element in this pair (i.e., full lyrics). The algorithm iterates through all matches and concatenates words from the start to the end index of each match into a string. Upon confirming that these lyrics haven't appeared before, it prints them along with their corresponding song title, incorporates them into the set of already printed lyrics; subsequently setting 'found_match' as True.

After all iterations if it conclude without a matching result, the system will print a message: "The abbreviation does not correspond to any lyrics or song title."

From a stylised point of view I ensured to insert space in my output to improve visibility and readability of the songs titles/lyrics.

Function : Main

```python
from reading_jsonfile import get_titles_lyrics
from user_intput import user_input
from extract_first_letters import extract_lyric_first_letters, extract_title_first_letters
from compare_input_to_data import compare_input_to_data

try:
    song_details = get_titles_lyrics()
    title_details = extract_title_first_letters(song_details)
    lyrics_details = extract_lyric_first_letters(song_details)
    continue_decoding = "YES"
    while continue_decoding.upper() == "YES":
        try:
            letters = user_input()
            if letters is not None:
                compare_input_to_data(letters)
        except ValueError as e:
            print(e)
            continue
        while True:
            try:
                continue_decoding = input("Do you want to decode another acronym? (yes/no): ")
                if continue_decoding.upper() not in ["YES", "NO"]:
                    raise ValueError("Invalid input")
                break
            except ValueError:
                print("Characters others than 'yes' or 'no' was typed.")
except Exception as e:
    print(f"An error occurred: {e}")
```

My programming approach emphasizes the active structuring of code into distinct scripts and functions, a strategy that notably enhances readability, maintainability, and reusability. By maintaining the main function as an independent script, I can keep my program's entry point separate; this division offers a clear overview of the flow within my application,thus facilitating understanding of the code.

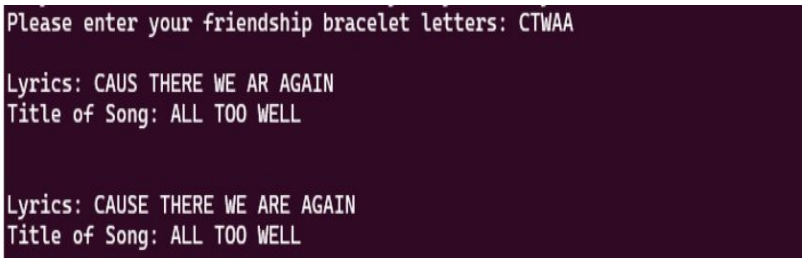
I made sure not to include the scrapping script in my main script to avoid f executing the scrapping every time a process that can consume significant time and resources; I perform the task once and stored the data. In this approach, I mitigate the risk of superfluous repetitive scraping processes and excessive requests.

I have also integrated an additional input/ouput interactive feature into my primary function, which augments user experience significantly. The program prompts a question to the user after each operation of decoding acronyms: "Would you like to decode another acronym? (yes/no)." The program initiates another decoding operation upon receiving a "YES" response from the user; alternatively, it gracefully concludes if the user responds with "NO". This feature ensure continuous and seamless operation flow maintains user engagement without necessitating a program restart.

Using a try block in my code offers a pivotal advantage: it gracefully handles unexpected errors or exceptions during program execution. Should an error arise within the try block,rather than abruptly halting the program, which could compromise user experience, it transfers control to the except block. Subsequently, this aidful piece of programming prints out an informative error message without interrupting overall flow. If user did type anything else then the ‘yes’ or ‘no’,it would raise a ValueError then it would print a message and re-prompt to ask if you wish to continue decoding.

Challenges and Improvement

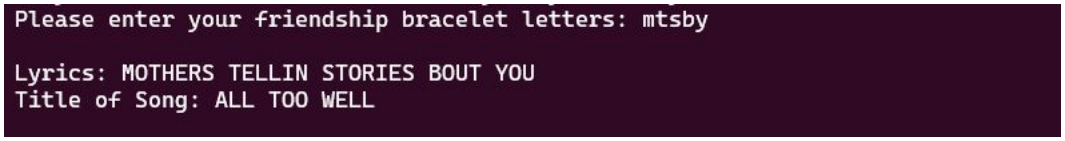
1. Data cleaning was a massive challenge due to the nature of scrapping the lyrics of songs some of the data being not as clean as possible eg. this is essentially the same lyrics but because of one letter difference it will print to the terminal twice as it is not considered a duplicate. Due to time constraints I honestly had to accept it. The purpose of the application was to decode and it is decoding even if it's printing "duplicates" in a sense.



Challenges and Improvement

2. I had to remove grammar marks from the dataset to match input and for the purpose of the decoder to work as explained in my function. Had I had more time I would have like work out a way for my input to still have the correct grammar.

Eg. terminal to be “your mother's telling stories 'bout you” instead of the below which had the grammar marks removed.



Favourite part

1. I can go to my taylor swift concert knowing I don't have to manually decode bracelet. I got a working application YAY
2. Code Hierarchy just clicked all of a sudden for me
3. I enjoyed learning more about bash scripts

```
Please enter your friendship bracelet letters: kiacpimlcilm

Lyrics: KARMA IS A CAT PURRING IN MY LAP CAUSE IT LOVES ME
Title of Song: KARMA
```