Midterm 1 - Fall 2024: Professor Vuduc Writes a Pop Song

Version 1.0.0

All of the header information is important. Please read it..

Topics number of exercises: This problem builds on your knowledge of `['string processing', 'nested data', 'sorting']`. It has **10** exercises numbered 0 to **9**. There are **18** available points. However to earn 100% the threshold is **13** points. (Therefore once you hit **13** points you can stop. There is no extra credit for exceeding this threshold.)

Exercise ordering: Each exercise builds logically on previous exercises but you may solve them in any order. That is if you can't solve an exercise you can still move on and try the next one. Use this to your advantage as the exercises are **not** necessarily ordered in terms of difficulty. Higher point values generally indicate more difficult exercises.

Demo cells: Code cells starting with the comment `### define demo inputs` load results from prior exercises applied to the entire data set and use those to build demo inputs. These must be run for subsequent demos to work properly but they do not affect the test cells. The data loaded in these cells may be rather large (at least in terms of human readability). You are free to print or otherwise use Python to explore them but we may not print them in the starter code.

Debugging your code: Right before each exercise test cell there is a block of text explaining the variables available to you for debugging. You may use these to test your code and can print/display them as needed (careful when printing large objects you may want to print the head or chunks of rows at a time).

Exercise point breakdown:

- Exercise 0 - : **1 FREE** point(s)
- Exercise 1 - : **1** point(s)
- Exercise 2 - : **2** point(s)
- Exercise 3 - : **1** point(s)
- Exercise 4 - : **3** point(s)
- Exercise 5 - : **2** point(s)
- Exercise 6 - : **2** point(s)
- Exercise 7 - : **2** point(s)
- Exercise 8 - : **3** point(s)
- Exercise 9 - : **1** point(s)

Final reminders:

- Submit after **every exercise**
- Review the generated grade report after you submit to see what errors were returned
- Stay calm, skip problems as needed and take short breaks at your leisure

Environment setup

Run the following cells to set up the problem.

In [1]:
```
%load_ext autoreload
%autoreload 2

import dill
import re
from pprint import pprint
from cse6040_devkit import plugins
```

The problem: Professor Vuduc wants to be a pop star, but needs your help!

Your overall task: Professor Vuduc wants to become a pop star, but he needs your help to write his new hit song! First, you'll need to analyze Spotify's Most Streamed Songs from 2023, and determine what attributes his song should contain. Then, you'll analyze lyrics from some of these most streamed artists. Finally, we will combine the results of this analysis to build a lyric generator capable of writing Professor Vuduc's new song!

The datasets:

- The first dataset is the metadata for Spotify's Most Streamed Songs from 2023. The Spotify dataset was sourced from here (https://www.kaggle.com/datasets/rajatsurana979/most-streamed-spotify-songs-2023?resource=download).
- The second dataset is the raw lyrics dataset. The lyrics were scraped from Genius (https://genius.com/) using their LyricsGenius (https://lyricsgenius.readthedocs.io/en/master/) Python client.

Run the cells below to load the data, and view samples of the two variables: `spotify_metadata` and `raw_lyrics`

Note: You might notice that some of the `raw_lyrics` differ slightly from the original song lyrics. This is because we are using the Kidz Bop version of the lyrics for each artist/song in `raw_lyrics`. Professor Vuduc wants to ensure the whole family can enjoy his new hit song!

Load Dataset 1: `spotify_metadata`

In [2]:
```
with open('resource/asnlib/publicdata/spotify_metadata.dill', 'rb') as fp:
    spotify_metadata = dill.load(fp)

print(f"=== Success: Loaded {len(spotify_metadata):,} Spotify song metadata re
cords. ===")
print(f"\nExample: Records 0 and 7:\n")
pprint([spotify_metadata[k] for k in [0, 7]])
```

```
=== Success: Loaded 701 Spotify song metadata records. ===

Example: Records 0 and 7:

[{'acousticness_%': '31',
  'artist_count': '2',
  'artist_name': 'Latto, Jung Kook',
  'bpm': '125',
  'danceability_%': '80',
  'energy_%': '83',
  'instrumentalness_%': '0',
  'key': 'B',
  'liveness_%': '8',
  'mode': 'Major',
  'released_year': '2023',
  'speechiness_%': '4',
  'streams': '141381703',
  'track_name': 'Seven',
  'valence_%': '89'},
 {'acousticness_%': '83',
  'artist_count': '1',
  'artist_name': 'David Kushner',
  'bpm': '130',
  'danceability_%': '51',
  'energy_%': '43',
  'instrumentalness_%': '0',
  'key': 'D',
  'liveness_%': '9',
  'mode': 'Minor',
  'released_year': '2023',
  'speechiness_%': '3',
  'streams': '387570742',
  'track_name': 'Daylight',
  'valence_%': '32'}]
```

Load Dataset 2: `raw_lyrics`

In [3]:
```
with open('resource/asnlib/publicdata/lyrics.dill', 'rb') as fp:
    raw_lyrics = dill.load(fp)

print(f"=== Success: Loaded song lyrics from {len(raw_lyrics):,} artists. ==
=")
print(f"=== Success: Loaded {len([song for song_dict in raw_lyrics.values() fo
r song in song_dict]):,} song lyrics total. ===")
print(f"\nExample: Harry Styles - 'As It Was' First 12 Lines:\n")
pprint({'Harry Styles': {'As It Was': raw_lyrics['Harry Styles']['As It Was']
[:12]}})
```

```
=== Success: Loaded song lyrics from 120 artists. ===
=== Success: Loaded 435 song lyrics total. ===

Example: Harry Styles - 'As It Was' First 12 Lines:

{'Harry Styles': {'As It Was': ["Holdin' me back",
                                "Gravity's holdin' me back",
                                'I want you to hold out the palm of your han
d',
                                "Why don't we leave it at that?",
                                "Nothin' to say",
                                'When everything gets in the way',
                                'Seems you cannot be replaced',
                                "And I'm the one who will stay, oh-oh-oh",
                                '',
                                "In this world, it's just us",
                                "You know it's not the same as it was",
                                "In this world, it's just us"]}}
```

Part 1: Analyzing Spotify Metadata

Exercise 0: (1 points) `spotify_metadata__FREE`

This is a free exercise!

The first dataset we will be working with is the metadata of the most streamed songs on Spotify in 2023: `spotify_metadata` is a list of dictionaries, where each dictionary contains the information for a single song.

Each dictionary contains the following keys, and all values are of the data type **string**:

- 'acousticness_%'
- 'artist_count'
- 'artist_name'
- 'bpm'
- 'danceability_%'
- 'energy_%'
- 'instrumentalness_%'
- 'key'
- 'liveness_%'
- 'mode'
- 'released_year'
- 'speechiness_%'
- 'streams'
- 'track_name'
- 'valence_%'

Please run the test cell below to collect your FREE point

In [4]:
```
### Test Cell - Exercise 0


print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Exercise 1: (1 points) `compute_song_stats`

Your task: Define `compute_song_stats` as follows: Compute the average BPM, danceability, and number of streams for the Spotify Top Songs of 2023.

Input: `spotify_metadata`: A list of dictionaries, as described in Exercise 0.

Return: A tuple containing the following in order: (`average_bpm`, `average_danceability`, `average_streams`)

Requirements:

- Compute the average `bpm`, average `danceability`, average `streams` for all songs in `spotify_metadata`
- Round each average to the nearest integer
- Format average `streams` to be a string with commas as the thousands separator. For example, the integer 1000 should be represented as '1,000'

Hint: This example https://stackoverflow.com/questions/1823058/how-to-print-a-number-using-commas-as-thousands-separators (https://stackoverflow.com/questions/1823058/how-to-print-a-number-using-commas-as-thousands-separators) may help with the formatting requirement

In [5]:
```
### Solution - Exercise 1
def compute_song_stats(spotify_metadata: list) -> tuple:
    ### BEGIN SOLUTION
    bpm = []
    dance = []
    streams = []

    for song in spotify_metadata:
        bpm.append(int(song['bpm']))
        dance.append(int(song['danceability_%']))
        streams.append(int(song['streams']))

    def mean(x):
        return sum(x)/len(x)

    def format_streams(x):
        return '{:,}'.format(round(x))

    return (round(mean(bpm)), round(mean(dance)), format_streams(mean(stream
s)))
    ### END SOLUTION

### Demo function call
song_stats_demo_input = spotify_metadata[:3]
print(compute_song_stats(song_stats_demo_input))
```

```
(118, 67, '138,367,321')
```

Example. A correct implementation should produce, for the demo, the following output:

```
(118, 67, '138,367,321')
```

The cell below will test your solution for compute_song_stats (exercise 1). The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. Any `key:value` pair in `original_input_vars` should also exist in `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [6]:
```
### Test Cell - Exercise 1

from cse6040_devkit.tester_fw.testers import Tester
from yaml import safe_load

with open('resource/asnlib/publicdata/assignment_config.yaml') as f:
    ex_conf = safe_load(f)['exercises']['compute_song_stats']['config']


ex_conf['func'] = compute_song_stats

tester = Tester(ex_conf, key=b'xH1i2Ha4qxJQ5O6vK9uj_3UQoel_h6vw4MwPpikJhzw=',
path='resource/asnlib/publicdata/')
for _ in range(90):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(ex_conf, key=b'RZvKu1zN_0kFp4jzi3eTiyjpuDiF9UH8tIfNb2fQcHw=',
path='resource/asnlib/publicdata/encrypted/')
for _ in range(10):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Exercise 2: (2 points) `find_songs_by_artist`

Your task: Define `find_songs_by_artist` as follows: Reorganize Spotify metadata into a dictionary of lists of tuples, containing the song information for each artist.

Input: `spotify_metadata`: A list of dictionaries, as described in Exercise 0.

Return: `songs_by_artist`: A dictionary of lists of tuples, where the artist is the key, and the value is a list of tuples of the form: (`track_name`, `streams`)

Requirements:

- Split the `artist_name` field as needed to handle multiple artists. Multiple artists are separated by commas
- Trim off any whitespace from the `artist_name` strings
- Convert `streams` to an integer, and sort the list of tuples for each artist by the number of streams in descending order

In [7]:
```
### Solution - Exercise 2
def find_songs_by_artist(spotify_metadata: list) -> dict:
    ### BEGIN SOLUTION
    from collections import defaultdict
    songs_by_artist_dict = defaultdict(list)

    for i in spotify_metadata:
        for j in i['artist_name'].split(","):
            tup = (i['track_name'], int(i['streams']))
            songs_by_artist_dict[j.strip()].append(tup)

    for (artist, tracks) in songs_by_artist_dict.items():
        songs_by_artist_dict[artist] = sorted(tracks, key =lambda x: -x[1])
    return dict(songs_by_artist_dict)
    ### END SOLUTION

### Demo function call
songs_by_artist_demo_input = [spotify_metadata[k] for k in [0, 41]]
pprint(find_songs_by_artist(songs_by_artist_demo_input))
```

```
{'BTS': [('Left and Right', 720434240)],
 'Charlie Puth': [('Left and Right', 720434240)],
 'Jung Kook': [('Left and Right', 720434240), ('Seven', 141381703)],
 'Latto': [('Seven', 141381703)]}
```

Example. A correct implementation should produce, for the demo, the following output:

```
{'BTS': [('Left and Right', 720434240)],
 'Charlie Puth': [('Left and Right', 720434240)],
 'Jung Kook': [('Left and Right', 720434240), ('Seven', 141381703)],
 'Latto': [('Seven', 141381703)]}
```

The cell below will test your solution for find_songs_by_artist (exercise 2). The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. Any `key:value` pair in `original_input_vars` should also exist in `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [8]:
```
### Test Cell - Exercise 2

from cse6040_devkit.tester_fw.testers import Tester
from yaml import safe_load

with open('resource/asnlib/publicdata/assignment_config.yaml') as f:
    ex_conf = safe_load(f)['exercises']['find_songs_by_artist']['config']


ex_conf['func'] = find_songs_by_artist

tester = Tester(ex_conf, key=b'LyTdGFBVK3zjO6u1Afo9Py7pkIGtCg0_OS8DIGKe05Q=',
path='resource/asnlib/publicdata/')
for _ in range(90):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(ex_conf, key=b'jmUUjFfLt21L1GTTN46o_MWZUXNVjjrApww0fb4QL0s=',
path='resource/asnlib/publicdata/encrypted/')
for _ in range(10):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Run me!

Whether your solution is working or not, run the following code cell, which will preload the results of Exercise 2 in the global variable, `songs_by_artist`.

In [9]:
```
with open('resource/asnlib/publicdata/top_songs.dill', 'rb') as fp:
    songs_by_artist = dill.load(fp)

print(f"=== Success: Loaded stats from {len(songs_by_artist):,} artists. ===")
```

```
=== Success: Loaded stats from 484 artists. ===
```

Exercise 3: (1 points) `discover_top_artists`

Your task: Define `discover_top_artists` as follows: Given the result of Exercise 2, return a list of the top X artists with the most songs in the Spotify metadata.

Input:

- `songs_by_artist`: A dictionary of lists of tuples, where the artist is the key, and the value is a list of tuples of the form: (`track_name`, `streams`)
- `X`: An integer representing the maximum number of tuples to return.

Return: `top_artists`: A list of X tuples of the form: (artist name, number of songs, number of total streams)

Requirements:

- Count the number of songs for each artist
- Sum the total number of streams for each artist
- Create list of tuples where each tuple corresponds to one artist: (artist name, number of songs, number of total streams)
- Sort this list in descending order by song count. If the song counts are the same, sort by total streams in descending order
- Return at most `X` tuples

In [10]:
```
### Solution - Exercise 3
def discover_top_artists(songs_by_artist: dict, X: int) -> list:
    ### BEGIN SOLUTION
    counts = []
    for (artist, tracks) in songs_by_artist.items():
        total_streams = sum([i[1] for i in tracks])
        counts.append((artist, len(tracks), total_streams))
    full_sorted = sorted(counts, key =lambda x: (-x[1], -x[2]))
    truncated = full_sorted[:X]

    return truncated
    ### END SOLUTION

### Demo function call
top_artists_demo_input = {k: songs_by_artist[k] for k in ['Latto', 'Jung Koo
k', 'Myke Towers']}
print(discover_top_artists(top_artists_demo_input, 2))
```

```
[('Jung Kook', 5, 1469963422), ('Latto', 1, 141381703)]
```

Example. A correct implementation should produce, for the demo, the following output:

```
[('Jung Kook', 5, 1469963422), ('Latto', 1, 141381703)]
```

The cell below will test your solution for discover_top_artists (exercise 3). The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. Any `key:value` pair in `original_input_vars` should also exist in `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [11]:
```
### Test Cell - Exercise 3

from cse6040_devkit.tester_fw.testers import Tester
from yaml import safe_load

with open('resource/asnlib/publicdata/assignment_config.yaml') as f:
    ex_conf = safe_load(f)['exercises']['discover_top_artists']['config']


ex_conf['func'] = discover_top_artists

tester = Tester(ex_conf, key=b'6oIUE7811jz51nFds0V_2Ya-FtZEQ6kDeqgrqEhv7Oo=',
path='resource/asnlib/publicdata/')
for _ in range(90):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(ex_conf, key=b'dD8-3MF7NAWbU6zKo-OC9C34NmHFXX99cFlb9ZvY2kg=',
path='resource/asnlib/publicdata/encrypted/')
for _ in range(10):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Part 2: Analyzing Lyrics

We will be working with data from `raw_lyrics` for the remainder of the exercises.

`raw_lyrics` is a dictionary of dictionaries of lists, where the outermost key is the artist name, the value is a dictionary where the keys are the song titles and their values are lists containing the raw lyric data. See below for an example of a single artist's entry in `raw_lyrics`:

In [12]:
```
pprint({'Doja Cat': raw_lyrics['Doja Cat']})
```

```
{'Doja Cat': {'Say So': ["Didn't like to know it, keep with me in the momen
t",
                         "I'd say it had I known it, why don't you say so?",
                         "Didn't even notice, no punches left to roll with",
                         'You got to keep me focused; you know it? Say so',
                         "Didn't like to know it, keep with me in the momen
t",
                         "I'd say it had I known it, why don't you say so?",
                         "Didn't even notice, no punches left to roll with",
                         'You got to keep me focused; you know it? Say so',
                         '',
                         "It's been a long time since you fell in love",
                         "You ain't coming out your shell, you ain't really "
                         'been yourself',
                         'Tell me, what must I do? (Do tell, my love)',
                         "'Cause luckily I'm good at reading",
                         "I wouldn't tell him, but he won't stop cheesin'",
                         'And we can dance all day around it',
                         "If you frontin', I'll be bouncing",
                         'If you know it, scream it, shout it, babe',
                         'Before I leave you dry',
                         '',
                         "Didn't like to know it, keep with me in the momen
t",
                         "I'd say it had I known it, why don't you say so?",
                         "Didn't even notice, no punches left to roll with",
                         'You got to keep me focused; you know it? Say so',
                         "Didn't like to know it, keep with me in the momen
t",
                         "I'd say it had I known it, why don't you say so?",
                         "Didn't even notice, no punches left to roll with",
                         'You got to keep me focused; you know it? Say so',
                         "Didn't like to know it, keep with me in the momen
t",
                         "I'd say it had I known it, why don't you say so?",
                         "Didn't even notice, no punches left to roll with",
                         'You got to keep me focused; you know it? Say so',
                         "Didn't like to know it, keep with me in the momen
t",
                         "I'd say it had I known it, why don't you say so?",
                         "Didn't even notice, no punches left to roll with",
                         'You got to keep me focused; you know it? Say so',
                         'You might also like',
                         'Ooh, ah-ha-ah-ah-ha-ah-ha-ah-ha',
                         'Ooh, ah-ha-ah-ah-ha-ah-ha-ah-ha']}}
```

Exercise 4: (3 points) `cleanse_lyrics`

Your task: Define `cleanse_lyrics` as follows: Given a list of lyrics for a single song, cleanse the text of each line, and return the cleansed list.

Input: `lyrics_list`: A list of lyrics for one song, where each element of the list corresponds to one line of lyrics in that song.

Return: `cleansed_lyrics_list`: A list of song lyrics, where the raw text is cleansed as outlined in the rules below.

Recommended Steps:

1. Combine the lyrics into one string, separated by newline characters **"\n"**
2. Make all characters lowercase
3. Replace any hyphens **"-"** with a single space **" "**
4. Remove any **non-alphabetic** characters, with the exception of any whitespace characters, single quote characters **"'"**, and parentheses **"()"**
5. Remove background vocals (any words inside of parentheses and the parentheses themselves). You may assume that any open parenthesis will eventually be followed by a closed parenthesis, though a backup vocal may span multiple lines. You may also assume there will be no nested parentheses.
6. Split the combined string with newline characters **"\n"** as the delimiter
7. Trim off any whitespace characters from the beginning and end of each line
8. Remove any empty lines

In [13]:
```
### Solution - Exercise 4
def cleanse_lyrics(lyrics_list: list) -> list:
    ### BEGIN SOLUTION
    import re
    cleansed_list = []

    lyrics = ('\n').join(lyrics_list).lower()

    lyrics_no_hyphens = re.sub(r'\-',' ', lyrics)
    pattern = re.compile(r'([a-zA-Z\'\s\)\(]*)')
    matches_list = pattern.findall(lyrics_no_hyphens)
    ##rejoin
    new_line = ''.join(matches_list)

    new_line = re.sub(r'\([a-zA-Z\'\s]*\)','', new_line)

    for i in new_line.split('\n'):
        if i.strip() != '':
            cleansed_list.append(i.strip())

    return cleansed_list
    ### END SOLUTION

### Demo function call
cleanse_lyrics_demo_input = raw_lyrics['Doja Cat']['Say So'][-27:-19] + raw_ly
rics['Doja Cat']['Say So'][-7:]
pprint(cleanse_lyrics(cleanse_lyrics_demo_input))
```

```
['tell me what must i do',
 "'cause luckily i'm good at reading",
 "i wouldn't tell him but he won't stop cheesin'",
 'and we can dance all day around it',
 "if you frontin' i'll be bouncing",
 'if you know it scream it shout it babe',
 'before i leave you dry',
 "didn't like to know it keep with me in the moment",
 "i'd say it had i known it why don't you say so",
 "didn't even notice no punches left to roll with",
 'you got to keep me focused you know it say so',
 'you might also like',
 'ooh ah ha ah ah ha ah ha ah ha',
 'ooh ah ha ah ah ha ah ha ah ha']
```

Example. A correct implementation should produce, for the demo, the following output:

```
['tell me what must i do',
 "'cause luckily i'm good at reading",
 "i wouldn't tell him but he won't stop cheesin'",
 'and we can dance all day around it',
 "if you frontin' i'll be bouncing",
 'if you know it scream it shout it babe',
 'before i leave you dry',
 "didn't like to know it keep with me in the moment",
 "i'd say it had i known it why don't you say so",
 "didn't even notice no punches left to roll with",
 'you got to keep me focused you know it say so',
 'you might also like',
 'ooh ah ha ah ah ha ah ha ah ha',
 'ooh ah ha ah ah ha ah ha ah ha']
```

The cell below will test your solution for cleanse_lyrics (exercise 4). The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. Any `key:value` pair in `original_input_vars` should also exist in `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [14]:
```
### Test Cell - Exercise 4

from cse6040_devkit.tester_fw.testers import Tester
from yaml import safe_load

with open('resource/asnlib/publicdata/assignment_config.yaml') as f:
    ex_conf = safe_load(f)['exercises']['cleanse_lyrics']['config']


ex_conf['func'] = cleanse_lyrics

tester = Tester(ex_conf, key=b'9x2HROuOnqJCQO021wsYDHF6s5EXXs4IA6GVAD-AB-k=',
path='resource/asnlib/publicdata/')
for _ in range(90):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(ex_conf, key=b'aoLS-BjlsgVtKbs67DFBBQwtFKMOSQDZVwlSxXm6d3I=',
path='resource/asnlib/publicdata/encrypted/')
for _ in range(10):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Run me!

Whether your solution is working or not, run the following code cell, which will preload the results of Exercise 4 in the global variable, `all_cleansed_lyrics`.

In [15]:
```
with open('resource/asnlib/publicdata/all_cleansed_lyrics.dill', 'rb') as fp:
    all_cleansed_lyrics = dill.load(fp)

print(f"=== Success: Loaded cleansed lyrics from {len(all_cleansed_lyrics):,}
artists. ===")
```

```
=== Success: Loaded cleansed lyrics from 120 artists. ===
```

Exercise 5: (2 points) `vibe_check`

Your task: Define `vibe_check` as follows: Given a list of cleansed lyrics, identify which words appear most frequently in a song, so that we can determine the "vibe" of the song.

To do this effectively, we should first remove any stop words from the lyrics before counting the occurrences of each word. Stop words are a set of words that are so commonly used in the English language that they carry little useful information to our analysis.

Input:

- `cleansed_lyrics_list`: A list of cleansed lyrics from a single song.
- `X`: An integer representing the maximum number of words to return.

Return: `top_vibes`: A set of up to the top X most common words found in the lyrics.

Requirements:

- Remove stopwords using the global variable `STOP_WORDS`
- Count occurrences of each word
- Return a set containing at most `X` common words. If the counts of two words are the same, sort by **length of the word** in descending order

Hint: Remember that sets in Python are unordered. Your result must contain the same words as the expected result, but the order may differ!

In [16]:
```
### Load our Stop Words:
with open('resource/asnlib/publicdata/stopwords.dill', 'rb') as fp:
    STOP_WORDS = dill.load(fp)

print(f"=== Success: Loaded {len(STOP_WORDS):,} stop words. ===")
```

```
=== Success: Loaded 174 stop words. ===
```

In [17]:
```
### Solution - Exercise 5
def vibe_check(cleansed_lyrics_list: list, X: int) -> set:
    ### BEGIN SOLUTION
    from collections import Counter

    ## Combine all lines
    full_lyrics  = ' '.join(cleansed_lyrics_list)

    words_list = full_lyrics.split()

    ## Remove stopwords
    words_list_no_stopwords = [i for i in words_list if i not in STOP_WORDS]

    ## Create counter and store
    counts_dict = Counter(words_list_no_stopwords)

    ## Sort items in Counter dictionary by count descending, then by length of
the word itself in descending order
    sorted_tuple_list = sorted(counts_dict.items(), key = lambda x: (-x[1], -l
en(x[0])))

    ## Grab 0th element of tuple for first X tuples, convert to set and return
    return set([word[0] for word in sorted_tuple_list[:X]])
    ### END SOLUTION

### Demo function call
vibe_check_demo_input = all_cleansed_lyrics['Taylor Swift']['Cruel Summer']
print(vibe_check(vibe_check_demo_input, 3))
```

```
{'cruel', 'oh', 'summer'}
```

Example. A correct implementation should produce, for the demo, the following output:

```
{'oh', 'summer', 'cruel'}
```

The cell below will test your solution for vibe_check (exercise 5). The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. Any `key:value` pair in `original_input_vars` should also exist in `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [18]:
```
### Test Cell - Exercise 5

from cse6040_devkit.tester_fw.testers import Tester
from yaml import safe_load

with open('resource/asnlib/publicdata/assignment_config.yaml') as f:
    ex_conf = safe_load(f)['exercises']['vibe_check']['config']


ex_conf['func'] = vibe_check

tester = Tester(ex_conf, key=b'aAJm0NmL6IXkjrs_VRLsb8ZU7tC-cbIjMxDJY-9Hpts=',
path='resource/asnlib/publicdata/')
for _ in range(90):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(ex_conf, key=b'NsRD6dzBYr1c94MXNcUcR4zH2oSfnyMG0qUIxa8gHFQ=',
path='resource/asnlib/publicdata/encrypted/')
for _ in range(10):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Exercise 6: (2 points) `generate_bigrams`

Your task: Define `generate_bigrams` as follows: Given a list of the cleansed lyrics, find the count of all word bigrams.

What is a bigram? A bigram is a pair of consecutive written units. In our exercise, we want to find the counts of pairs of consecutive **words** in **each line of lyrics**.

Input: `cleansed_lyrics_list`: A list of cleansed lyrics from a single song.

Return: `bigrams_dict`: A dictionary in which the key is a tuple (first_word, second_word), and the value is the count of the number of times that bigram appears in the lyrics.

Requirements:

- For each line of lyrics, find the count of all word bigrams
- Generate a dictionary where the key is a tuple of the bigram, and its value is the count of the number of times that bigram appeared in the lyrics

Example: For the line 'you might also like', the bigrams would be ('you', 'might'), ('might', 'also'), and ('also', 'like').

In [19]:
```
### Solution - Exercise 6
def generate_bigrams(cleansed_lyrics_list: list) -> dict:
    ### BEGIN SOLUTION
    from collections import Counter

    # Create empty counter dict to hold running total bigram count
    total_bigrams_count = Counter()

    # Iterate over each line in lyrics list
    for line in cleansed_lyrics_list:
        # Split string of words into list using default whitespace delimiter
        list_of_words_in_line = line.split()
        # Zip together the list of words with the next words. Count and add to
total Counter
        total_bigrams_count += Counter(zip(list_of_words_in_line,list_of_words
_in_line[1:]))

    return dict(total_bigrams_count)
    ### END SOLUTION

### Demo function call
bigrams_demo_input = all_cleansed_lyrics['Doja Cat']['Say So'][-3:]
pprint(generate_bigrams(bigrams_demo_input))
```

```
{('ah', 'ah'): 2,
 ('ah', 'ha'): 8,
 ('also', 'like'): 1,
 ('ha', 'ah'): 6,
 ('might', 'also'): 1,
 ('ooh', 'ah'): 2,
 ('you', 'might'): 1}
```

Example. A correct implementation should produce, for the demo, the following output:

```
{('ah', 'ah'): 2,
 ('ah', 'ha'): 8,
 ('also', 'like'): 1,
 ('ha', 'ah'): 6,
 ('might', 'also'): 1,
 ('ooh', 'ah'): 2,
 ('you', 'might'): 1}
```

The cell below will test your solution for generate_bigrams (exercise 6). The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. Any `key:value` pair in `original_input_vars` should also exist in `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [20]:
```
### Test Cell - Exercise 6

from cse6040_devkit.tester_fw.testers import Tester
from yaml import safe_load

with open('resource/asnlib/publicdata/assignment_config.yaml') as f:
    ex_conf = safe_load(f)['exercises']['generate_bigrams']['config']


ex_conf['func'] = generate_bigrams

tester = Tester(ex_conf, key=b'0SJ3lw8EW4pH2sJ0vG2hYTV27GP_7FmJHNnV3MYGTsc=',
path='resource/asnlib/publicdata/')
for _ in range(90):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(ex_conf, key=b'cMXfpdHn0BKjrpp_EQ7BFE2mKQ97j-5r7XXaZ41Ca8Q=',
path='resource/asnlib/publicdata/encrypted/')
for _ in range(10):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Run me!

Whether your solution is working or not, run the following code cell, which will preload the results of Exercise 6 in the global variable, `bigrams_dict`.

In [21]:
```
with open('resource/asnlib/publicdata/all_bigrams.dill', 'rb') as fp:
    bigrams_dict = dill.load(fp)

print(f"=== Success: Loaded {len(bigrams_dict):,} bigrams. ===")
```

```
=== Success: Loaded 29,403 bigrams. ===
```

Exercise 7: (2 points) `rhyme_time`

Your task: Define `rhyme_time` as follows: Given a list of cleansed lyrics, generate a rhyming dictionary using the last word in each line of lyrics.

Input: `cleansed_lyrics_list`: A list of cleansed lyrics from a single song.

Return: `rhyming_dict`: A dictionary in which the key is the last word of a lyric line, and its value is a set of all of the last words found in `cleansed_lyrics_list` that rhyme with the key.

Requirements:

- Check if any of the last words in the list of lyrics rhyme with each other. Use only the provided helper function `rhyme_checker` to determine if two words rhyme
- The helper function `rhyme_checker` returns `True` if the two words provided rhyme and `False` otherwise
- If any words rhyme, add the rhyming words to the dictionary symmetrically for each word. For example, in the demo below, note that both `'are': {'car'}` and `'car': {'are'}` appear

In [22]:
```
### Load our Rhyming Data
with open('resource/asnlib/publicdata/rhyming_dict.dill', 'rb') as fp:
    rhyme_lookup = dill.load(fp)
with open('resource/asnlib/publicdata/lookup.dill', 'rb') as fp:
    lookup = dill.load(fp)
```

In [23]:
```
### Helper Function
def rhyme_checker(word1, word2):
    return word1 in plugins.rhymes(word2, lookup, rhyme_lookup)

### Solution - Exercise 7
def rhyme_time(cleansed_lyrics_list: list) -> dict:
    ### BEGIN SOLUTION
    from collections import defaultdict

    rhyming_dict = defaultdict(set)
    last_words = []
    for line in cleansed_lyrics_list:
        ## extract last word in every line
        last_words.append(line.split()[-1])
    for i in last_words:
        for j in last_words:
            if rhyme_checker(i,j):
                rhyming_dict[i].add(j)
                rhyming_dict[j].add(i)

    return dict(rhyming_dict)
    ### END SOLUTION

### Demo function call
rhyming_demo_input = all_cleansed_lyrics['Taylor Swift']['Cruel Summer']
pprint(rhyme_time(rhyming_demo_input))
```

```
{'are': {'car'},
 'below': {'oh'},
 'car': {'are'},
 'fate': {'gate'},
 'gate': {'fate'},
 'lying': {'trying'},
 'oh': {'below'},
 'true': {'you'},
 'trying': {'lying'},
 'you': {'true'}}
```

Example. A correct implementation should produce, for the demo, the following output:

```
{'are': {'car'},
 'below': {'oh'},
 'car': {'are'},
 'fate': {'gate'},
 'gate': {'fate'},
 'lying': {'trying'},
 'oh': {'below'},
 'true': {'you'},
 'trying': {'lying'},
 'you': {'true'}}
```

The cell below will test your solution for rhyme_time (exercise 7). The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. Any `key:value` pair in `original_input_vars` should also exist in `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [24]:
```
### Test Cell - Exercise 7

from cse6040_devkit.tester_fw.testers import Tester
from yaml import safe_load

with open('resource/asnlib/publicdata/assignment_config.yaml') as f:
    ex_conf = safe_load(f)['exercises']['rhyme_time']['config']


ex_conf['func'] = rhyme_time

tester = Tester(ex_conf, key=b'ayf9kq6by6ehuJv9J_-MRoQ7ae8BwPXEwout_w2hu4o=',
path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(ex_conf, key=b'JHEVcw1EvSNmqG0zbE0D5tm6xhnCGCbAUEdmF9SAgh4=',
path='resource/asnlib/publicdata/encrypted/')
for _ in range(5):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Exercise 8: (3 points) `count_syllables`

Your task: Define `count_syllables` as follows: Given a set of words, find the number of syllables in each word.

Input: `set_of_words`: A set of words, such as {'are', 'car', 'trying'}

Return: `syllable_dict`: A dictionary in which the keys are the words found in `set_of_words`, and the value is the number of syllables in that word.

Requirements: To determine the number of syllables in a word:

1. If the first letter of the word is a letter within `vowels_no_y`, add 1
2. If there is a letter within `consonants` followed immediately by a letter within `vowels`, add 1 for each occurrence
3. If there are **3 or more** of any of the letters within `vowels` consecutively, add 1 for each occurrence. For instance, 'uoyy' would be considered a valid match
4. Now check to see if the word ends with an `'e'`. If so, subtract 1, unless the word ends with `'le'` and the preceding letter is a letter within `consonants`, then do nothing
5. Every word should have at least 1 syllable. If it has 0 syllables, add a syllable

Create a dictionary containing the word as a key and the number of syllables in that word as its value.

Hint: While not required, Regular Expressions (https://www.w3schools.com/python/python_regex.asp) might be helpful to use for Steps 1-4!

Examples:

- The word 'quiet' contains 2 syllables, as 'qu' adds 1 syllable in Step 2, and 'uie' adds 1 syllable in Step 3.
- The word 'the' contains 1 syllable, as 'he' adds 1 syllable in Step 2.
- The word 'you' contains 1 syllable, as 'you' adds 1 syllable in Step 3.
- The word 'trouble' contains 2 syllables, as 'ro' adds 1 syllable in Step 2, 'le' adds 1 syllable in Step 2, and because it ends in 'le' and 'b' is a consonant we do not subtract 1.
- The word 'stale' contains 1 syllable, as 'ta' adds 1 syllable in Step 2, 'le' adds 1 syllable in Step 2, but you subtract 1 syllable in Step 4 because the word ends in 'le' and is not preceded by a consonant.
- The word 'irritate' contains 3 syllables, as 'i' adds 1 syllable in Step 1, 'ri' adds 1 syllable in Step 2, 'ta' adds 1 syllable in Step 2, 'te' adds 1 syllable in Step 2, and 'e' subtracts 1 syllable from our count in Step 4.

In [25]:
```
### Solution - Exercise 8
def count_syllables(set_of_words: set) -> dict:
    vowels_no_y = 'aeiou'
    vowels = 'aeiouy'
    consonants = 'bcdfghjklmnpqrstvwxz'

    ### BEGIN SOLUTION
    syllable_dict = {}

    # Iterate over words in set_of_words
    for word in set_of_words:

        # Initialize syllable count for each word to be 0
        word_syllable_count = 0

        # Step 1: Check if first letter is vowel_no_y, if so, increase word_sy
llable_count by 1
        starting_vowel = re.findall('^[aeiou]', word)
        word_syllable_count += len(starting_vowel)

        # Step 2: Check if consonant followed by vowel, if so, increase word_s
yllable_count by 1 for each occurrence
        vowel_after_consonant = re.findall('[bcdfghjklmnpqrstvwxz][aeiouy]', w
ord)
        word_syllable_count += len(vowel_after_consonant)

        # Step 3: Check if there are 3 or more vowels in a row, if so, increas
e word_syllable_count by 1 for each occurrence
        three_vowels = re.findall('[aeiouy]{3,}', word)
        word_syllable_count += len(three_vowels)

        # Step 4: If last letter in word is an e, decrease syllable count by
1, unless ends in consonant + le
        ending_le = re.findall('[bcdfghjklmnpqrstvwxz]le$', word)
        if len(ending_le) == 0 and word[-1] == 'e':
            word_syllable_count -= 1

        # Step 5: Syllable count for a word must be at least 1
        if word_syllable_count <= 0:
            word_syllable_count = 1

        syllable_dict[word] = word_syllable_count

    return syllable_dict
    ### END SOLUTION

### Demo function call
syllables_demo_input = {'queue', 'luckily', 'quiet', 'the', 'a', 'you', 'troub
le', 'irritate', 'stale'}
pprint(count_syllables(syllables_demo_input))
```

```
{'a': 1,
 'irritate': 3,
 'luckily': 3,
 'queue': 1,
 'quiet': 2,
 'stale': 1,
 'the': 1,
 'trouble': 2,
 'you': 1}
```

Example. A correct implementation should produce, for the demo, the following output:

```
{'a': 1,
 'irritate': 3,
 'luckily': 3,
 'queue': 1,
 'quiet': 2,
 'stale': 1,
 'the': 1,
 'trouble': 2,
 'you': 1}
```

The cell below will test your solution for count_syllables (exercise 8). The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. Any `key:value` pair in `original_input_vars` should also exist in `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [26]:
```
### Test Cell - Exercise 8

from cse6040_devkit.tester_fw.testers import Tester
from yaml import safe_load

with open('resource/asnlib/publicdata/assignment_config.yaml') as f:
    ex_conf = safe_load(f)['exercises']['count_syllables']['config']


ex_conf['func'] = count_syllables

tester = Tester(ex_conf, key=b'QGWv7NoU-PUzYjlptlEfeSBYFBQNaTeePcOXZAPcFbA=',
path='resource/asnlib/publicdata/')
for _ in range(90):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(ex_conf, key=b'v95v3ukvgG0eVvB0APHrnKdjIzvPt5vMI1ILiH-At2U=',
path='resource/asnlib/publicdata/encrypted/')
for _ in range(10):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Exercise 9: (1 points) `build_markov_process`

Your task: Define `build_markov_process` as follows: Using the result from Exercise 6, generate a Markov process so our lyric generator can select the next word in a lyric with probabilities matching those found in our lyric dataset. Before attempting this exercise, make sure you have loaded the global variable `bigrams_dict` located in the 'Run Me' code cell following Exercise 6.

Input: `bigrams_dict`: A dictionary containing bigram keys that are tuples (first_word, second_word), with values that are the count of the number of times that bigram appears in the lyrics.

Return: `markov_dict`: A dictionary of lists, in which the key is the first word, and the value is a list containing all potential second words.

Requirements:

- Create `markov_dict` in which the keys are the first words from `bigrams_dict`, and the values are a list of the second words in that bigram duplicated the number of times specified by the count for that bigram

Example: For input `{('first', 'second'): 2, ('first', 'other'): 1}`, your result would be: `{'first': ['second', 'second', 'other']}`

In [28]:
```
### Solution - Exercise 9
def build_markov_process(bigrams_dict: dict) -> dict:
    ### BEGIN SOLUTION
    from collections import defaultdict

    final_dict = defaultdict(list)

    for bigram_tuple, count in bigrams_dict.items():
        initial_word = bigram_tuple[0]
        next_word = bigram_tuple[1]
        next_word_count_list = [next_word] * count
        final_dict[initial_word].extend(next_word_count_list)
    return dict(final_dict)
    ### END SOLUTION

### Demo function call
markov_demo_input = {k: bigrams_dict[k] for k in [('like', 'ah'), ('like', 'wh
at'), ('ah', 'he'), ('he', 'got')]}
pprint(build_markov_process(markov_demo_input))
```

```
{'ah': ['he'],
 'he': ['got', 'got', 'got'],
 'like': ['ah', 'ah', 'ah', 'ah', 'what', 'what', 'what', 'what']}
```

Example. A correct implementation should produce, for the demo, the following output:

```
{'ah': ['he'], 'he': ['got', 'got', 'got'], 'like': ['ah', 'ah', 'ah', 'ah', 'wha
t', 'what', 'what', 'what']}
```

The cell below will test your solution for build_markov_process (exercise 9). The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. Any `key:value` pair in `original_input_vars` should also exist in `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [29]:
```
### Test Cell - Exercise 9

from cse6040_devkit.tester_fw.testers import Tester
from yaml import safe_load

with open('resource/asnlib/publicdata/assignment_config.yaml') as f:
    ex_conf = safe_load(f)['exercises']['build_markov_process']['config']


ex_conf['func'] = plugins.postprocess_sort_dict(build_markov_process)

tester = Tester(ex_conf, key=b'TfnCIOMcUBY0m-_81gdDHjykus0D9WgVS6gMasPLb_E=',
path='resource/asnlib/publicdata/')
for _ in range(90):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(ex_conf, key=b'aF5zVnTWaGlJX22D3S1IWXHxaMw-ZjPWzx9HzTo8NFE=',
path='resource/asnlib/publicdata/encrypted/')
for _ in range(10):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS

print('Passed! Please submit.')
```

```
Passed! Please submit.
```

Fin

If you have made it this far, congratulations! You are done. Please submit your exam!

The remainder of this notebook combines the work you have completed above to build a simple lyric generator and writes Professor Vuduc's new song.

Epilogue: It's Time to Generate Professor Vuduc's Song!

```
## Pre-processing Lyric Data:

# Step 0: Create one large list containing all song lyrics
all_lyrics_list = [line for song_dict in raw_lyrics.values() for lyrics in song_dic
t.values() for line in lyrics]

# Step 1: Cleanse lyrics of all songs (using Exercise 4)
all_cleansed_lyrics = cleanse_lyrics(all_lyrics_list)

# Step 2: Generate markov process from bigrams of all cleansed lyrics (using Exerci
ses 6 and 9)
all_bigrams_dict = generate_bigrams(all_cleansed_lyrics)
all_markov_process = build_markov_process(all_bigrams_dict)

# Step 3: Generate rhyming words dictionary for all cleansed lyrics (using Exercise
7) (load this as it takes a long time to run)
with open('resource/asnlib/publicdata/epilogue_rhyming_dict.dill', 'rb') as fp:
    all_songs_rhyming_dict = dill.load(fp)

# Step 4: Generate syllables dictionary for all words in cleansed lyrics (using Exe
rcise 8)
all_words = {word for line in all_cleansed_lyrics for word in line.split()}
all_words_syllables_dict = count_syllables(all_words)

# Step 5: Find most common starting words of each lyric line (optional, using Exerc
ise 4 result)
with open('resource/asnlib/publicdata/epilogue_top_starting_words.dill', 'rb') as f
p:
    top_starting_words_sorted = dill.load(fp)

## ----------------------------------------------------------------------------
------------
```

```
## Create a function to generate a single line of lyrics:

# Step 6: Create generate_lyric function with inputs: starting word of each line of
lyrics, desired number of syllables per line, and the word to rhyme end word with
(optional):
#         1. Use glabal variable 'all_words_syllables_dict' to count syllables as w
e go
#         2. Generate next_word using global variable 'all_markov_process'
#         3. Select final word of lyric line from global variable 'all_songs_rhymin
g_dict'
from random import sample

def generate_lyric(first_word_of_lyric, desired_syllables, rhyming_word=None):
    total_syllable_count = all_words_syllables_dict[first_word_of_lyric]
    lyric = first_word_of_lyric
    next_word = first_word_of_lyric

    # If we were provided a rhyming word as input, find a word that rhymes with it
to become the end of our next line of lyrics
    if rhyming_word:
        new_rhyming_word = sample(all_songs_rhyming_dict[rhyming_word], 1)[0]
        total_syllable_count += all_words_syllables_dict[new_rhyming_word]

    # While our number of syllables for the line of lyrics is less than the desired
number of syllables, keep generating words
    while total_syllable_count < desired_syllables:
        prior_word = next_word
        next_word = sample(all_markov_process[prior_word], 1)[0]
        tries = 0
        while next_word not in all_markov_process and tries < 50:
            next_word = sample(all_markov_process[prior_word], 1)[0]
            tries += 1
        if next_word not in all_markov_process:
            next_word = sample(list(all_markov_process.keys()), 1)[0]
        lyric = lyric + ' ' + next_word
        total_syllable_count += all_words_syllables_dict[next_word]

    # If a rhyming word was not provided as input, randomly choose an ending rhymin
g word
    if rhyming_word is None:
        if next_word not in all_songs_rhyming_dict:
            final_word = sample(list(all_songs_rhyming_dict.keys()), 1)[0]
            lyric = lyric + ' ' + final_word
            total_syllable_count += all_words_syllables_dict[final_word]
            return lyric, final_word, min(total_syllable_count, 10)
        else:
            return lyric, next_word, min(total_syllable_count, 10)
    else:
```

```
        lyric =  lyric + ' ' + new_rhyming_word
        return lyric, None, min(total_syllable_count, 10)


## ----------------------------------------------------------------------------
------------

# Step 7: Repeatedly call Step 6's generate_lyric function and add to final list of
lyrics

# Choose Song Structure:
    # 2 verses of 6 lines each
    # Chorus of 8 lines
    # 2 verses of 6 lines each
    # Same chorus of 8 lines again

verse_lyrics = []
chorus_lyrics = ['[Chorus:]']
syllable_count = 10

for j in range(4):
    verse_count = j+1
    verse_lyrics.append(f'[Verse {verse_count}:]')
    verse_starting_words = sample(top_starting_words_sorted, 6)
    for i, starting_word in enumerate(verse_starting_words):
        if i % 2:
            one_lyric_line, rhyming_word, syllable_count = generate_lyric(starting_
word, syllable_count, rhyming_word)
            verse_lyrics.append(one_lyric_line)
        else:
            one_lyric_line, rhyming_word, syllable_count = generate_lyric(starting_
word, syllable_count)
            verse_lyrics.append(one_lyric_line)
    verse_lyrics.append('\n')

chorus_starting_words = sample(top_starting_words_sorted, 8)
for i, starting_word in enumerate(chorus_starting_words):
    if i % 2:
        one_lyric_line, rhyming_word, syllable_count = generate_lyric(starting_wor
d, syllable_count, rhyming_word)
        chorus_lyrics.append(one_lyric_line)
    else:
        one_lyric_line, rhyming_word, syllable_count = generate_lyric(starting_wor
d, syllable_count)
        chorus_lyrics.append(one_lyric_line)

song_lyrics = verse_lyrics[:16] + chorus_lyrics + verse_lyrics[15:] + chorus_lyrics
song_lyrics_no_titles = verse_lyrics[1:7] + verse_lyrics[9:15] + chorus_lyrics[1:]
+ verse_lyrics[18:23] + verse_lyrics[25:31] + chorus_lyrics[1:]
```

```
## ------------------------------------------------------------------------
------------

# Step 8: Run vibe check on list of generated lyrics to choose a song title (using
Exercise 5)
vibe_analysis = vibe_check(song_lyrics_no_titles, 5)
print('Vibe of the song: ', vibe_analysis, '\n')

## ------------------------------------------------------------------------
------------

# Step 9: Join lyric lines with newline characters and return full string of the so
ng
final_song = '\n'.join(song_lyrics)
print(final_song)
```

Song Vibe:

```
{'oh', 'see', 'baby', 'get', 'like'}
```

Song Lyrics:

[Verse 1:]
don't you say never fade away with california gurls we're free
let's hope with me trippin' oh oh yeah me
you're tired of my heart is be another
we gotta gotta know i'm giving brother
i've been movin' so just need to replace
baby my head still breathing fire fire face

[Verse 2:]
it's not fazed only want from all eyes
it was you might also like you see skies
all these dreams come back time we were right to
i'll be without ya i don't get achoo
and i got nothing on my enemy your hand
this love right here drippin' off and i'm sand

[Chorus:]
hey i've been a thing for the way that chico nice
if the way your friends talk to find other twice
when you're beautiful liar bad blood hey yeah it's so
baby that's caught up higher over yo
and uh huh you you get our very special with you
got fake people you get you like nah nah do
i'll never see it takes you ever
i'm on get a little bit of forever

[Verse 3:]
yeah yeah yeah i'm coming down together
we are full of the applause applause weather
'cause i cry me go crazy what makes you
what people hatin' say the ice cream for knew
you better now i feel nothin' happens when
just wanna talk to forget her again

[Verse 4:]
ooh oh oh oh oh ooh i just like baby
it's not gonna walk that it's true no maybe
now i'm 'bout it again to fall as you
i was long time on these tears me through oooh
it didn't come kick him it that could see you
she was not around the rhythm and this two

[Chorus:]
hey i've been a thing for the way that chico nice
if the way your friends talk to find other twice
when you're beautiful liar bad blood hey yeah it's so
baby that's caught up higher over yo
and uh huh you you get our very special with you
got fake people you get you like nah nah do
i'll never see it takes you ever
i'm on get a little bit of forever