CS 120 (Fall 21): Introduction to Computer Programming II

Long Project #2

due at 5pm, **Wed** 8 Sep 2021

1 Overview

People have long been interested in figuring out similarities between two pieces of music. There are apps to identify tunes and duplicate music; people claim that the theme from the Harry Potter movies was copied from a work by a 19th century French compose; and there have been lots of lawsuits about music plagiarism. This raises an obvious question: can we write code to figure out how similar two pieces of music are?

1.1 Our Strategy

To make the problem manageable, we'll make a big simplifying assumption: we'll treat a melody as a simple sequence of notes - ignoring chords or harmonies. Moreover, we'll ignore the fact that you could play the same melody in two different keys. We'll simply look for **identical** sequences, note for note.

Of course, we don't expect to see two songs that are entirely identical. So what we'll look for, instead, are little strings of notes, somwhere in the songs, which are identical. The longer the string of notes, the more interesting; the more matches we find, the more similar we will say the songs are.

Our goal, then, is to compute a "similarity score" for two songs; the higher the score, the greater the similarity. A score of 0.0 means that there is nothing in common; a score of 1.0 means that they are as similar as can be.

Then, by scanning through a set of songs, and comparing the first song to all the rest, we will report which two songs are most similar: that is, the pair that has the highest similarity score.

2 Two Types of Problems

In this project, you will be turning in two files. The first one, `utils.py`, will contain a set of functions that we have defined for you: you must implement exactly what the spec requires. We will have testcases which check these functions.

You will also implement a complete program, named `music_compare.py` . This program should import your `utils.py`, and use the functions inside to help you calculate the similarity of various songs. We will have some testcases which test this program as well.

Make sure that your various utility functions are working properly before you worry about the complete program; it's hard to debug a program when its building blocks are broken.

3 Computing Similarity

There are **many** ways that we might compare the similarity of two songs, but we had to choose one for this project, so that it was possible to grade your work. The algorithm we chose was as follows:

- First, the user chooses a certain length, which we'll call N. (They give this to your program as an input.)
- Second, you will iterate through both songs, and slice out of them all of the sequences of length N.
- Third, you will collect the sequences, from each song, into a set. (Sets are useful because they remove duplicates, we'll explain below.)
- Fourth, you will ask Python how many items are in common between the two sets - this tells you how many small phrases the songs share.
- Fifth, you will calculate the "Jaccard index" (see below) to generate a value, between 0.0 and 1.0, which represents how similar the two songs are.

4 Doing Many Comparisons

The algorithm above shows you how to compare two arbitrary songs. Your program will do this repeatedly. Specifically, it will read in all of the songs from the file, and compare the first one to all of the others. It will report the result of each such comparison. At the end, it will also report the song which was most similar to the first song in the file.

5 Input File Format

(Melody listings adapted from `https://noobnotes.net/`)

You an input file. Each input file contains a list of songs. Each song specification spans two lines, with a blank line following. The first line provides information about the song; the second gives the notes of the song.

Examples:

```
@ 033 I'm a little teapot.
C D E F G ^C A ^C G F F F E E D D D C C D E F G ^C A ^C G ^C A G G F E D C

@ 052 Never be the same. Camila Cabello, 2017.
E G G E G G G E A G G G G G G G E A G G G G G G G E A G G G G G
```

You may assume that the file is always in the following format. (You are not expected to confirm that this is the case.)

- No comments to ignore
- The first song will begin on the first line (no leading blank line)
- Each song will have exactly two lines: an "info" line, and then a melody line.
- There will always be a single blank line between songs (although there might not be a blank line after the last song)
- All "info" lines will start with @, followed by a space, then a number written in decimal, then (optionally) one or more spaces, followed by some additional text.
- The notes of the various songs will be separated by spaces.

  (While notes often are pretty simple - a single character - don't assume that this is true! Sometimes, they include multiple characters.)

6 Output Format

For the required functions, your code should not print out **anything;** instead, they return values to their callers.

For the complete program, you will first give two prompts:

```
file:
n:
```

The first that you read will be the name of the file to open, which contains the music to compare; the second is an integer.

You will first print out a list of all of the songs in the file, followed by the details of each comparison; at the end, you will print out information about which song was most similar to the first one.

(Note: When printing out the sets of notes at the end, you should sort each set, so that its order is predictable. Otherwise, you may not pass the autograder.)

(spec continues on next page)

EXAMPLE:
(All of the melodies and comparison numbers are made up. Pay attention to the **format,** not the actual values.)

```
--- SONG LIST ---
id=10 info="I Love You, You Love Me.  Barney the Purple Dinosaur." notes=['A', 'B', 'C',
id=123 info="This is the info line." notes=['A', 'B', 'C']
id=456 info="This has a lot more info on this song." notes=['A', 'C#', 'E']
id=12 info="November Rain.  Guns 'n' Roses." notes=['A', 'B', 'C', 'G', 'A', 'B', 'C',

--- COMPARISONS ---
id_a=10 id_b=123 similarity=0.1234
id_a=10 id_b=456 similarity=0.1234
id_a=10 id_b=12 similarity=0.997

--- RESULT ---
Most similar songs:
  I Love You, You Love Me.  Barney the Purple Dinosaur.
  November Rain.  Guns 'n' Roses.

  ids: 10
  ids: 12

  Melodies:
    A B C D A B C D
    A B C G A B C G

  Set 1
    A B C
    B C D
    C D A
    D A B

  Set 2
    A B C
    B C G
    C G A
    G A C
```

7 Algorithms

7.1 Slicing

We have chosen a fairly simple way to compare songs. Our first step is to slice out of a given song all of the sequences of a certain length. (In the full program, the length we are extracting is the input `N`, which you will read from the keyboard .)

For example, suppose we have the string of notes and `N=3`.

```
A B C A B C A C D A B C
```

We can find many slices of length three in the sequence:

```
A B C A B C A C D A B C
-----------------------
A B C
  B C A
    C A B
      A B C
        B C A
          C A B
            A B C
              B C A
                C A B
                  A B C
```

All slices that we collect must be the **full length** - so our last possible slice, in the example above, is `A,B,C` at the **end** of the melody.

7.2 Sets

NOTE: See the Appendix, at the end of this spec for a description of Python's `set` class.)

In any given melody, some of the slices are likely to be duplicates of each other. For this algorithm, we choose to eliminate the duplicates; we will only keep one copy of each possible slice.

To do this, we use Python's `set` class; if you try to insert a duplicate value into a `set`, Python automatically ignores you and doesn't change anything. (The major downside of a `set` is that it is unsorted. So you'll have to sort the set - turning it into an array in the process - if you want to have data in a predictable order.)

In our example above, we end up with only 3 different slices:

```
A B C
B C A
C A B
```

(spec continues on next page)

7.3 Jaccard Index

The Jaccard Index is a way to compare two sets. The idea is that, if the sets have nothing in common, then the Jaccard index should be zero. If they are exactly the same, the it should be 1. Given two sets `A,B` the Jaccard Index is defined as

$$similarity(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

How do you perform intersection (the upward-facing operator) and union (the downward facing one) on Python sets? Check out the documentation!

`https://docs.python.org/3/tutorial/datastructures.html#sets`

7.4 Summary

In summary, to compare two songs, you will:

- Do slicing to get the list of all sequences from each melody, at a certain chosen length.
- Convert both to sets.
- Compute the Jaccard Index between the sets.

8 Required Functions

The file `utils.py` must declare the following functions:

8.1 `read_file(fobj)`

This function takes one paramemter - an already-open file object - and returns the music information within. The file will be in the format described above; the returned data must be a list of tuples (in the same order the songs were in the file). Each tuple will have exactly 3 parts:

- The song number (as listed in the input file), converted to an integer.
- Any **additional** text about a song (all found on the “info” line, above the melody.

  This **must not** include the @, nor the number of the song, nor the whitespace before any additional text on that line.
- The actual nodes of the melody, as a list of strings.

8.2 get_slices(data, n)

Generates **all** of the slices for a given list, at a given length. It takes two parameters: the data (which can be any sort of sliciable data, includeing strings, lists, etc.), and a length.

You must return all of the possible slices in a list. As we discussed above, you **must not** include short slices - so, for instance, if the requested length is 4, then you **must not** include the 3-element slice, the 2-element slice, and the 1-element slice which would arise from starting the slice very close to the end.

Duplicates are legal for this function - if the input has two regions which have the same sequence of values, then return a slice for each one. Return them in the same order that they occured inside the input data.

EXAMPLE:

```
data = [2,3,5,7,11,13]
slices = get_slices(data, 4)

RETURNS:
[ [2,3,5,7], [3,5,7,11], [5,7,11,13] ]
```

8.3 compare_sets(a,b)

Given two sets, computes the Jaccard Index between them, and returns it.

While (in this program), we generally assume that the members of these sets are lists of notes from a melody, you program should not assume that; write a general-purpose function, which can compare sets made up of anything. Our testcases will test your code on a variety of different data types.

8.4 compare_melodies(m1, m2, n)

This function takes three parameters. The first two are melodies; the third is the N value that you are going to use to perform the comparison between the two songs. Each melody is encoded as a list of short strings, where each string represents a note.

This function must compute the similarity of the two songs, and return it.

9 Turning in Your Solution

You must turn in your code using GradeScope.

(spec continues on next page)

10 Appendix: Python's `set`

Python includes a standard `set` class, which models a mathematical set: it stores values, but does not allow duplicates, and the values are stored in no particular order.

To create a new empty set, use the following syntax:

```
a = set()
```

To create a set using one or more values, use syntax that looks a lot like a dictionary:

```
b = { 1,2,3,4 }
```

To add a new value to a set, use the `add()` method. If the value is already in the set, this operation will complete silently but nothing will be changed:

```
c = set()
c.add("asdf")
c.add("jkl")
c.add("asdf")      # NOP
```

To get the size of a set, use the `len()` function:

```
d = { ... }
size = len(d)
```

To iterate over the values of a set, use a `for` loop:

```
e = { ... }
for val in e:
```

To sort the values of a set (this will return an array), call the `sorted()` function. This is often combined with a for loop:

```
f = { ... }
for val in sorted(f):
```

To remove a value from a set, call `remove()`:

```
g = { "asdf", "jkl" }
g.remove("asdf")
```

To convert an existing data structure (array, tuple, etc) into a set, use the `set()` constructor:

```
data = [ ... ]
h = set(data)
```

Note that, because the values in a set are not in any particular order, indices are meaningless; therefore, you cannot use the `[]` operator on a set (even if it kind of looks a little like a dictionary).

Also note that (just like keys in a dictionary) the only values that can be stored in a set are immutable types (integers, strings, tuples, etc.). This is because a set, just like a dictionary is based on a Hash Table. (We will probably do a quick overview of Hash Tables at the end of the semester, but it's not required material right now.)

11 Acknowledgements

Thanks to Saumya Debray for this assignment!