Lab #3 **Making Music With Strings**

Overview: In this lab, you will write various functions to manipulate strings using for-loops. To spice up the action, for this lab each string will represent the musical notes of a short song, and your functions will perform various musical transformations to a given song.

Getting Started: For this lab, we will use the accompanying Python code in the file `MakeSong.py` posted on Google Classroom. This accompanying code will allow you to easily create a WAV audio file (see https://en.wikipedia.org/wiki/WAV) that you can play, which will correspond to the musical notes in a given string representing a song/tune.

To facilitate getting started, I have also provided a starter `lab3_music_with_strings.py` program in which the remainder of your programming work will go. You will note that the program follows the recommendations of using a `main` function and nothing declared in the global scope.

- From Google Classroom, download and save both `MakeSong.py` and the starter `lab3_music_with_strings.py` into a folder called `lab3`.
- Upload MakeSong.py and the starter `lab3_music_with_strings.py` to Google Colab. Running the starter file should create a new song file named `cello.wav` in Google Colab.

Inside the given `lab3_music_with_strings.py`, I have provided four song strings as starters, for *Mary Had A Little Lamb*, *Twinkle Twinkle Little Star*, *Smoke On The Water* (Deep Purple), and *Suite No. 1 for Cello* (J.S. Bach). We want you to have fun with this assignment. Feel free to create your own song strings!

IMPORTANT: All of your coding work below will go inside the `lab3_music_with_strings.py` source code file. Remember that function definitions should go at the top of your program, below the import statement(s) but above `main`. Only the single call to `main()` should be in the global scope. Make sure to follow good style throughout your work — good variable and parameters names, good use of vertical and horizontal space, good commenting including docstrings, and type hinting.

Part I:

1. Write a fruitful function named `reverseSong` having a single parameter named `song`. Use the algorithm from class to build a new song string from scratch — use a for-loop to iterate through the given song string parameter in reverse, appending note-by-note to your new string. Return your new string at the end. (For this assignment, you are not permitted to use Python "string slicing".)

   Back in `main`, include at least three appropriate simple (not necessarily musically interesting) tests of your fruitful function, clearly outputting the actual result and the expected result. Make sure to use the `printTest` function. Below is one good example for how to structure such a test:

   ```
   song     = "abcde"
   result   = reverse_song(song)
   expected = "edcba"
   printTest(f"reverse_song({repr(song)})", result, expected)
   ```

   Note that the test clearly indicates in printed output (a) what is being tested, (b) what the actual result is, and (c) what the expected result is.

   Beyond these tests, experiment with reversing more interesting songs (whether the ones I provided, or your own creations) and then creating a WAV file of the reversed songs using `MakeSong` as demonstrated in the provided `lab3_music_with_strings.py`. Let me know what interesting tunes you come up with!

2. Write a fruitful function named `raiseOctave` having a single parameter. Given a song string consisting of (lowercase) notes in the lower octave, this function must return a new string in which the notes have been raised (uppercased) to the upper octave.

   Use the algorithm from class to build a new song string from scratch: iterate through the given song note-by-note, and append to your new string an uppercased version of each note, returning the upper-octave string at the end. Note that, inside your for-loop, an expression like that shown below will do the right thing to uppercase the ith note in a song:

   ```
   song[i].upper()  # convert the note at index i to uppercase
   ```

   IMPORTANT: What is shown above is an expression only. Remember to append the result of that expression to your new string. Also, for this task, you must use `upper()` on each character, not on the entire provided song string.

   Back in `main`, include at least three appropriate simple (not necessarily musically interesting) tests of your fruitful function, clearly outputting the actual result and the expected result. Also then play around with more interesting musical songs, and listen to the results.

3. Write a fruitful function named `lowerOctave` having a single parameter. Given a song string consisting of (uppercase) notes in the upper octave, this function must return a new string in which the notes have been lowered (lowercased) to the lower octave character-by-character, similar to `raiseOctave` above.

   Back in `main`, include at least three appropriate simple (not necessarily musically interesting) tests of your fruitful function, clearly outputting the actual result and the expected result. Also then play around with more interesting musical songs, and listen to the results.

Part II: Random Songs

1. Write a fruitful function named `raiseOctavePart` having three parameters named `song`, `start`, and `stop`. Given a song string consisting of (at least some) lowercase notes in the lower octave, this function must return a newly-built string in which only the notes between indices `start` and `stop` both inclusive have been raised to the upper octave (see `raiseOctave` above). For example, a call inside `main` to this function as follows:

   ```
   song   = "ABcdeFG"
   #         0123456 -- string indices
   raised = raiseOctavePart(song, 2, 4)
   ```

   should result in the string `"ABCDEFG"` being stored in the `raised` variable.

   For this task, you must use the algorithm from class to build a new string from scratch, iterating note-by-note (string slicing is not permitted). You will need multiple separate (not nested) for-loops to solve this problem — how many? As always, work on a solution on paper before attempting to write any code.

   Back in `main`, include at least three appropriate simple (not necessarily musically interesting) tests of your fruitful function, clearly outputting the actual result and the expected result. Also then play around with more interesting musical songs, and listen to the results.

2. Write a fruitful function named `lowerOctavePart` having three parameters named `song`, `start`, and `stop`. Given a song string consisting of (at least some) uppercase notes in the upper octave, this function must return a newly-built string in which only the notes between indices `start` and `stop` both inclusive have been lowered to the lower octave (see `lowerOctave` above).

   Back in `main`, include at least three appropriate simple (not necessarily musically interesting) tests of your fruitful function, clearly outputting the actual result and the expected result. Also then play around with more interesting musical songs, and listen to the results.

3. Write a fruitful function named `randomSong` having three parameters named `notes`, `min_notes`, and `max_notes`. This function must build from scratch and then return a new string consisting of notes chosen at random from the given `notes` string. The length of the newly-built string must also be chosen at random between `min_notes` and `max_notes`, both inclusive.

   Recall that the following expressions (you must use the expressions meaningfully inside your code, not as stand-alone "ghost" expressions!) will respectively (a) choose an integer at random between `min_notes` and `max_notes` inclusive, and (b) choose an index at random between 0 and the last index in `notes` inclusive:

   ```
   random.randint(min_notes, max_notes)    # this is only an expression!

   random.randint(0, len(notes) - 1)       # this is only an expression!
   ```

   Remember to `import random` at the top of your program, as shown in class.

   Because the songs are produced at random, you can't know what the expected result will be... or can you? If you go into the Python interpreter at the terminal (or equivalently use Colab), you can use an initial seed to the random number generator to determine what the result of your program should be for a given set of `notes`, `min_notes`, and `max_notes` parameters, as demonstrated below.

```
>>> import random
>>> notes = "abcdefg"  # don't include sharps -- see below
>>> random.seed(5551212)
>>> num_notes = random.randint(5, 10)
>>> print(num_notes)
5
>>> for i in range(num_notes):
...   index = random.randint(0, len(notes) - 1)
...   print(notes[index])
...
c
f
e
e
e
```

so that you will know the expected result of a call to `randomSong("abcdefg", 5, 10)` should be `"cfeee"`, so long as you call `random.seed(5551212)` before your call to `random_song`. Back in `main`, include at least three appropriate tests of your fruitful function in which you produce random songs.

Note: Because you are choosing a single character at a time from `notes`, this approach won't work if you're including sharps (which consist of two characters representing the sharped note) — so don't include any sharps in the string you pass as an argument to `notes` here.

Optional Challenge: If you're looking for an additional challenge, you may choose to implement this function (not required). Write a fruitful function named `splice_song` having five parameters: `song1`, `splice_where`, `song2`, `start2`, and `end2` This function must build from scractch and then return a new string subject to the following. Take the notes from `song2`, starting at the note with index *start2* inclusive and ending at the note with index `end2` inclusive, and insert those notes just before the note at index `splice_where` in `song1`. The rest of `song1` must persist. (Again, you are not permitted to use string slicing here.) You may presume that the values given in `splice_where`, `start2`, and `end2` are valid values in the context.

As an example, the following test should result in the song string `"cdeEFfgab"` being stored in the `spliced` variable.

```
song_one = "cdefgab"
# indices: 0123456
song_two = "CDEFGAB"
spliced = splice_song(song_one, 3, song_two, 2, 3)
# spliced should contain "cdeEFfgab"
```

Back in `main`, include at least four appropriate tests of your function, clearly indicating the produced and expected results.

Submitting: When finished, simply upload in Google Classroom your `lab3.py`.