

CS 3430: Python & Perl
Assignment 7: Transcribing Bee Symphonies

Vladimir Kulyukin
Department of Computer Science
Utah State University

1. Learning Objectives

1. Py & PL Regular Expressions
2. Py & PL String Manipulation
3. Py & PL File Processing

2. Programming Music

Recall that in Assignment 5 you processed temperature logs collected by an electronic beehive monitor. Another data type collected by the monitor is audio. There are various ways in which we can process audio samples of natural phenomena. For example, we can create a frequency spectrogram and analyze its features or process it with various audio filters to discover which filters respond best, etc. We can train a neural network to recognize the presence of a specific feature, e.g., a call of a blue whale.

Another way to analyze natural sounds is to detect musical notes in them. For example, we can try to detect the presence/absence of the A440 piano notes (any other instrument would do just fine so long its frequencies have a tuning standard) in the audio samples taken inside the beehive. Assuming that our detection is numerically stable, i.e., detects similar notes in similar audio samples, we can use those notes to analyze and, possibly, predict various bee behavior patterns. After all, a sequence of notes is just another mathematical time series and all those wonderful techniques discovered by such giants as Fourier, Riemann, and Newton can be put to productive use.

The files that you will be working with are generated by a Java package I wrote for detecting the A440 piano notes in wav audio samples (<https://github.com/VKEDCO/AudioTrials>). While it is not yet as numerically stable as I would like it to be and the experiments are ongoing, I have a modestly reasonable degree of confidence in its performance. Of course, gaining more confidence means testing it on terabytes, not megabytes, as has been the case so far.

One of the classes in this package takes a wav file and generates a LOGO music file for the LOGO Music Player (LMP). To the best of my knowledge, the LMP is a module of most LOGO IDEs (https://en.wikipedia.org/wiki/Logo_%28programming_language%29). This module allows you to program, play, and save music in various formats. Here is a simple text file that the LMP can play. The numbers at the beginning of each line are not part of LOGO syntax. I wrote them for ease of reference. If you are not familiar with letter or sheet music transcription, do not worry – this assignment does not require you to be a composer or a transcriber.

```
1. PLAY [i48 t333 v127]
2. PLAY [[A0# B0 C1 ]]
3. PLAY [[A0# B0 C1 C1# D1 ]]
4. PLAY [[B0 C1 C1# D1 D1# G1 G1# ]]
5. PLAY [C3#]
6. PLAY [[A0# B0 C1 C1# D1 ]]
```

PLAY is a LOGO function. The remainder of each line is the arguments to this function in square brackets. For example, the first line specifies the instrument (i48), tempo (t333) and volume (v127). The first line always looks like this in an LMP file. It is the so-called tune-up instruction. You may assume that it will be the same in all LPM files. The other lines contain chords or notes. Some lines have arguments enclosed in a single pair of matching brackets, e.g., the 1st and 5th lines, whereas others have double matching brackets. What is the difference? The tune-up instruction always has single brackets. This is just part of the syntax. When another line has a single pair of brackets, e.g., 5th line, it means that the chord consists of only one note. Double brackets mean that a chord consists of multiple notes. One of the nice things about musical programming in LOGO (and similar IDEs) is that you can write and play chords consisting of more than ten notes. It is not humanly possible to play those chords – not enough fingers. So your program ends up sounding like an orchestra, after a fashion.

3. Bee Symphony Transcription

Suppose that there is a directory of small wav files, each of which has a very short duration, i.e., 15 seconds or so. This directory may, for example, cover two weeks of audio sampling inside a beehive. Suppose that there is a tool we can use to convert those files into LPM format at various frequencies. If we want to detect significant patterns, we need longer (much longer!) series. Hence, the problem of merging shorter LPM files into one large file, which is the programmatic objective of this assignment. If you are not interested in music, do not worry - the techniques you will learn in this assignment generalize far beyond computer music. This is a classical problem in many big data pattern recognition projects – given a whole bunch of shorter files (samples), merge them into a big file by using some domain- and problem-specific criteria. What does the result of this merging sound like? I uploaded a wma file called **BeeSymphony_13_18_05jul2015_22050**. This file represents a musical rendition of what one of the beehive sounded like on 07/05/2015 when the detected chords were transcribed into an LMP file at a frequency of 22050 Hz and played in the LMP as a symphony orchestra. If you know sheet music, Figure 1 gives you a chunk of it in base cliff. I used ScoreCloud Studio (<http://scorecloud.com/>) to generate this score.

13_18_05jul2015_22050.mid

LyricistComposer

1. Acoustic Grand Piano

1. 3. 5. 7. 10.

Figure 1. ScoreCloud's Output on LMP's Wav File Converted to MIDI

Back to file processing though. The **hw07.zip** contains three subfolders: **pl_test_data**, **py_test_data**, and **real_data**. The **pl_test_data** and **py_test_data** contain the same files. I created two directories with the same data so that you can work on your Py and PL solutions in parallel if you want. Debug your PL and PY implementations before running them on **real_data**. The **real_data** folder contains a couple of raw wav files before they are converted into LPM format. They are there to enrich your background experience. They should be left alone during processing. **Figure 2** shows the contents of the directory **pl_test_data**.









 2015-08-15_15-41-32_5512_logo	10/24/2015 5:59 PM	TXT File	1 KB
 2015-08-15_15-41-32_11025_logo	10/24/2015 5:59 PM	TXT File	1 KB
 2015-08-15_15-41-32_22050_logo	10/24/2015 6:00 PM	TXT File	1 KB
 2015-08-15_15-41-32_44100_logo	10/23/2015 3:25 PM	TXT File	1 KB
 2015-08-15_16-01-32_5512_logo	10/24/2015 6:00 PM	TXT File	1 KB
 2015-08-15_16-01-32_11025_logo	10/24/2015 6:00 PM	TXT File	1 KB
 2015-08-15_16-01-32_22050_logo	10/24/2015 6:00 PM	TXT File	1 KB
 2015-08-15_16-01-32_44100_logo	10/23/2015 4:05 PM	TXT File	1 KB

Figure 2. Contents of PL_TEST_DATA

Each file name has the following format: **<date>_<time>_<frequency>_logo.txt**, where **<date>** is a hyphenated date in the form **year-month-day**, **<time>** is a hyphenated time in the form **hour-minutes-seconds** and **<freq>** consists of several digits (at least 4 at most 6) specifying the frequency at which the original wav was sampled to detect the notes.

3.1 Designing Regular Expressions for Filtering File Names for a Specific Frequency

Design regular expressions that test if a specific file name (e.g., **2015-08-15_15-41-32_5512_logo.txt**) is a legal LPM file name. Your regular expressions should use groups to retrieve the date, time, and frequency from the match. I deliberately do not specify how many regexes you need to define but you have to define at least one. You can combine everything into one expression or define four regexps, i.e., one for each frequency – 5512, 11025, 22050, and 44100.

3.2 Merging All Files with a Given Frequency into One File

Write a Py function and PL subroutine **merge_bee_music_file_for_freq** that takes three command line parameters: frequency, a directory with the LPM files and the name of the file where all LPM files will be merged sequentially as they are processed. If you save your implementation in **bee_music.py** and **bee_music.pl**, the command line calls look as follows:

```
> bee_music.pl 44100 pl_test_data/ pl_test_data/merged_44100.txt
> bee_music.py 44100 pl_test_data/ pl_test_data/merged_44100.txt
```

The first command line argument is frequency, i.e., 44100, the second is the directory with LPM files, i.e., **pl_test_data/**, and the third argument is the name of the merged file, i.e., **pl_test_data/merged_44100.txt**.

The function/sub **merge_bee_music_file_for_freq** opens a directory specified in the 2nd argument, gets a list of files in that directory, filters only those files whose names have a particular frequency (specified in the 1st argument), and copies PLAY instructions from each file sequentially into the merged file.

Let us consider an example. The **py_test_data** folder contains two 44100 frequency files: **2015-08-15_15-41-32_44100_logo.txt** and **2015-08-15_16-01-32_44100_logo.txt**. Their contents are given below, one after the other.

```
PLAY [i48 t333 v127]
PLAY [[B2 C3# D3 D3# ]]
PLAY [[A1 A1# B1 D1# E1 F1 F1# G1 A2 B2 F2# G2 C3 C3# D3 D3# ]]
PLAY [[C1# D1 D1# B2 C3# D3 D3# ]]
PLAY [[C3# D3 D3# ]]
```

```
PLAY [i48 t333 v127]
PLAY [[C3 C3# D3 ]]
PLAY [[B2 C3 C3# D3 ]]
PLAY [[C3 C3# D3 ]]
PLAY [[C3 C3# D3 ]]
```

Suppose you execute

```
>bee_music.py 44100 py_test_data/ py_test_data/merged_44100.txt
```

Then the contents of **py_test_data/merged_44100.txt** should be as follows:

```
PLAY [i48 t333 v127]
PLAY [[B2 C3# D3 D3# ]]
PLAY [[A1 A1# B1 D1# E1 F1 F1# G1 A2 B2 F2# G2 C3 C3# D3 D3# ]]
PLAY [[C1# D1 D1# B2 C3# D3 D3# ]]
PLAY [[C3# D3 D3# ]]
PLAY [[C3 C3# D3 ]]
PLAY [[B2 C3 C3# D3 ]]
PLAY [[C3 C3# D3 ]]
PLAY [[C3 C3# D3 ]]
```

Notice that the tune-up instruction of the second file, i.e., **PLAY [i48 t333 v127]**, is not included in the merged file. This is because in a legal LPM file there should be only one tune-up instruction. Obviously, if the directory contains more than two files for a specific frequency, like in the **real_data** folder, the tune-up instructions of the 2nd, 3rd and so files should not be included in the merged file. One tune-up per merged file!

3.2 Stripping PLAY Off

Implement a Python function and PL subroutine **gen_no_play_file(file_path)** that takes a path to an LPM File and creates a new file in the same directory that ends with **_logo_no_play.txt** where it saves only the chords and notes. For example, suppose this function is called on **pl_test_data/2015-08-15_15-41-32_44100_logo.txt**, whose contents are

```
PLAY [i48 t333 v127]
PLAY [[B2 C3# D3 D3# ]]
PLAY [[A1 A1# B1 D1# E1 F1 F1# G1 A2 B2 F2# G2 C3 C3# D3 D3# ]]
PLAY [[C1# D1 D1# B2 C3# D3 D3# ]]
PLAY [[C3# D3 D3# ]]
```

The function creates a new file **pl_test_data/2015-08-15_15-41-32_44100_logo_no_play.txt** whose contents are

```
B2 C3# D3 D3#
A1 A1# B1 D1# E1 F1 F1# G1 A2 B2 F2# G2 C3 C3# D3 D3#
C1# D1 D1# B2 C3# D3 D3#
C3# D3 D3#
```

In other words, the **no_play** file contains only the chords. It does not contain the tune-up instruction or the **PLAY** function calls. This type of file processing is very helpful if we want to compute frequency statistics of individual notes over a specific period of time.

Now integrate a call to **gen_no_play_file** to your implementation of **merge_bee_music_file_for_freq** so that as the **no_play** files are generated as the files for a specific **freq** are processed.

4. Coding Tips

I uploaded two files, **bee_music.py** and **bee_music.pl**, that you can use as templates for this assignment. They contain all the imports that you should need. Here is how, for example, you can use list comprehension and regexp in Py to get the files that match a specific regexp in a directory:

```
from os import listdir
from os.path import isfile, join
import re
import sys
```

```
dirfiles = filter(lambda x: re.match(pat, x) != None, [f for f in listdir(drcty) if isfile(join(drcty, f)) ])
```

Same thing in PL:

```
use strict;
use warnings;
use FileHandle;

sub grep_files {
    my ($dir, $pat) = @_;
    opendir(my $dir_handle, $dir) or die $!;
    my @files = grep { $_ =~ /$pat/ } readdir($dir_handle);
    closedir($dir_handle);
    return \@files;
}
```

Review the file processing code samples in Lecture 16 for examples of how to open, close, read from, and write to files. For this assignment you may assume that all your files are text so do not feel compelled to do byte-level file processing. Test your code on **pl_test_data** and **py_test_data**, then run it on **real_data**.

5. What to Submit

Save your Py solution in **bee_music.py** and your PL solution in **bee_music.pl** and submit both files in Canvas. Please do not give the files you submit other names, like **hw07.pl**. We are trying to automate grading with scripts as much as possible so consistent naming really helps.

Happy Hacking!