pippi Documentation

Release 1.0

Erik Schoster

January 21, 2015

Contents

Contents:

CHAPTER 1

Background

Pippi takes advantage of a few features of CPython:

- Doing string manipulations with python's internal C methods (like *string.join()*) is fast.
- The python standard library includes the *audioop* module which accepts audio as a binary string and returns the same. It is also fast.
- There's a handy *wave* module in the standard library, which makes importing and exporting PCM wave data simple.

Data format

Internally, all sound in pippi is stored and manipulated as **binary strings**.

What?

Python's approach to working with binary data is to use its string data type as a wrapper. To get a feel for this, lets first look at the structure of the type of binary audio data we're trying to represent.

Signed 16 bit PCM audio has these characteristics:

Each frame of audio represents an instantanious value corresponding to a position the speaker cones will take when sent to our computer's digital-to-analog converter. *PCM* stands for Pulse Code Modulation - the modulation part refers to the filtering needed to correct for aliasing distortion that would occur if an analog speaker simply jumped from value to value over time, rather than somehow smoothly interpolating between these values.

It is conventional to use signed 16 bit integers to represent an instantanious speaker cone position - this is also the format CD audio takes.

A signed integer means that instead of having a range between zero and some positive number, it has a range between some negative number and some positive number.

This is great for representing audio - at zero the speaker cone is at rest, at max it is pushed as far out as it can go, and at min it is pulled as far in as it can go.

The number of bits in the integer dictate the size of the number it is possible to represent.

A 16 bit integer can represent *2^16* discrete values - or *65,536* total values.

That means a signed integer will use about half of those possible values as positives, and half as negatives.

Half of *2^16* is *2^15*, or *32,768*. Because we need to account for a zero value, the range of our signed 16 bit integer is actually *-2^15* to *2^15 - 1*. Or *-32,768* to *32,767*.

The potential size of the integer - or the number of discrete values it can represent - corresponds to the possible dynamic range that can be represented in the audio recording. More values mean more subtle differences between loud sounds and soft sounds, and everything in between. [Bhob Rainey has a wonderful writeup on why this is something to pay

attention to.](http://bhobrainey.wordpress.com/2010/08/04/selected-occasions-of-handsome-deceit/) (Also his music rules, so be sure to check it out.)

All that said, it's fairly accepted that 16 bit audio can represent differences in loudness that comes close to the limit our brains can distinguish. Supporting higher bit rates in pippi is on the list of to dos, but only because that extra dynamic resolution becomes useful when you're transforming very quiet sounds, or sounds with a very limited dynamic range.

So, we could just work with lists of python integers, but doing operations in pure python can get pretty slow - especially when a system will quickly grow to working with minutes and hours of audio. By relying on the fast C backend for string manipulation and basic DSP, performance is actually pretty good.

Instead we represent each integer as a python string, and when doing synthesis, use the *struct* module to pack our integers into binary strings.

To turn the python integer *32,767* into a binary string, we can give *struct.pack* a format argument and it will convert the number into the correct binary string.

```
>>> import struct
>>> struct.pack('<h', 32767)
'\xff\x7f'
```

While it looks like we just got back an eight character string, this is actually a two character string, where the leading *x* is the way python indicates that the next two characters should be read as a hex value.

So if we have *44,100* frames of a single channel of 16 bit audio, internally we'd have a string whose length is actually twice that - *88,200* characters. With two channels, our string will have *176,400* characters.

One convenience pippi provides is a way to prevent you from accidentally splitting a sound in the middle of a two-character word, which will instantly turn your audio into a brand new Merzbow track.

If you have a 10 frame 2 channel sound (in the below example, silence) and want to grab the last 5 frames, you could do:

```
>>> import struct
>>> sound = struct.pack('<h', 0) * 2 * 10
>>> len(sound)
40
>>> sound = sound[:5 * 2 * 2]
>>> len(sound)
20
```

Five frames of stereo 16 bit audio represented as a string will have a length of 20 characters.

You may see how this could get annoying, fast. And an off-by-one error will produce Merzbow forthwith.

With pippi, to do the same, we use the cut method:

```
>>> from pippi import dsp
>>> dsp.flen(sound)
10
>>> sound = dsp.cut(sound, 5, 5)
>>> dsp.flen(sound)
5
```

Using the same silence 10 frames from the earlier example, we can check the actual length with *dsp.flen()*. (Which is short for 'frame length' or 'length in frames')

To do the cut, *dsp.cut()* accepts three params: first, the binary string to cut from, next the offset in frames where the cut should start, and third the length of the cut in frames.

Summary

Part of what pippi provides is a wrapper to working with python binary strings. This is actually a very handy thing. That's just a small part of the library though. Next we'll talk about doing basic synthesis with pippi, and using some of its waveshape generators for both audio and control data.

CHAPTER 2

Console quick start

Note: the pippi console currently depends on ALSA, and is therefore linux-only. The core dsp system is however fully cross platform.

Pippi only supports python 2.7.x at the moment, so verify that's what you're using

    $ python –version

The pippi console runs in the current directory, and uses any correctly formatted instrument scripts it finds as voices.

First, create a very simple instrument script:

```
# beep.py

from pippi import dsp

def play(ctl):
    # Default args to tone produce a 1 second long sinewave at 440hz
    out = dsp.tone()

    # Reduce the volume of the beep by 90%
    out = dsp.amp(out, 0.1)

    # Pad the output with 0.5 seconds of silence
    out = dsp.pad(out, '', dsp.stf(0.5))

    return out
```

Start pippi:

```
$ pippi
```

Run the example generator script:

```
^_- play beep
```

The ‘‘^_- ‘‘ is just the cheeky prompt for the pippi console.

The command `play` can be shorted to `p` - it must be followed by the filename of an instrument script in the same directory.

Note: everything below is outdated - the console is currently being rewritten!

While the instrument plays, open the file ‘example.py’ in the ‘orc’ directory with your favorite text editor and find the line that reads `freq = tune.ntf('a', octave=2)`.

Try changing `'a'` to `'e'` or `'f#'`. Or maybe try changing `octave=2` to `octave=4` or `octave=10`. Or find `sine2pi` and try changing it to `tri`, `impulse`, `vary` or `hann`.

Back in the pippi console, type `i` to get a list of the currently playing voices:

```
^_- i
01 gen: EX dev: default bpm:120.0
```

Change the global bpm:

```
^_- bpm 110
```

Stop the currently running voice - we give the `s` command a param `1` because that's the id of the voice we'd like to stop.:

```
^_- s 1
```

Start a group of 10 voices with the example generator:

```
^_- group 10 -- ex re
```

Pippi just sends each voice stream to alsa for summing, so we get some interesting distortion effects because every sound is playing very loudly.

Have each voice choose a random volume on each iteration. Find the appropriate line and change it to read:

```
volume = P(voice_id, 'volume', default=dsp.rand(0, 20)) / 100.0
```

Above we're trying to read a parameter passed in at the pippi console to set the volume. Since we didn't give the generator any volume param when we started our voices, it uses the default value passed as the third argument to `P` which in this case is a random number between 0 and 100, updated on each iteration.

Try setting the 5th voice's volume to 50. We can use the `u` or update command to change a running voice's parameters as it plays.:

```
^_- u 5 v:50
```

Now lets make the rhythm a bit more interesting. Find the line where a value is assigned to `beat` and update it to read:

```
beat = dsp.bpm2frames(bpm) / dsp.randint(1, 4)
```

Vary the chosen octave:

```
freq = tune.ntf('a', octave=dsp.randint(1, 5))
```

Randomly choose from a given set of pitches:

```
freq = tune.ntf(dsp.randchoose(['a', 'c#', 'f#'], octave=dsp.randint(1, 5)))
```

Apply an amplitute envelope to the beep. After line 34 (where we set `out` to our beep sound) and before line 37 (where `out` is passed into `dsp.pad` and silence is appended to the end) try adding:

```
out = dsp.env(out, 'sine')
```

Each voice begins playing as soon as rendering is finished, and the last iteration has also finished playing. To force voice playback to quantize to the master bpm, we can give any voice the `qu` command.

To update every voice spawned from the example generator, adding the `qu` command, type:

```
^_- uu ex qu
```

To stop every voice currently playing (but allow each iteration to play though before stopping):

```
^_- ss
```

Check out the documentation for pippi for more code examples you can use in generator scripts. I also have a small but growing collection of instruments and recipes for pippi in my hcj.py repo. Anti-copywrite. Do whatever you would like with this software.

dsp.**alias**(*snd*, *passthru=False*, *envelope=None*, *split_size=0*)
A simple time domain bitcrush-like effect.

The sound is cut into blocks between 1 and 64 frames in size if split_size is zero, otherwise split_size is the number of frames in each block.

Every other block is discarded, and each remaining block is repeated in place.

Set passthru to True to return the sound without processing. (Can be useful when processing grains in list comprehensions.)

By default, a random amplitude envelope is also applied to the final sound.

dsp.**bln**(*length*, *low=3000.0*, *high=7100.0*, *wform='sine2pi'*)
Time-domain band-limited (citation needed) noise generator.

Generates a series of single cycles of a given wavetype at random frequences within the supplied range.

Sounds nice & warm, if you ask me.

dsp.**breakpoint**(*values*, *size=512*)
Takes a list of values, or a pair of wavetable types and values, and builds an interpolated list of points between each value using the wavetable type. Default table type is linear.

dsp.**cache**(*s=''*, *clear=False*)
Simple write() wrapper to create and clear cache audio

dsp.**env**(*audio_string*, *wavetype='sine'*, *fullres=False*, *highval=1.0*, *lowval=0.0*, *wtype=0*, *amp=1.0*, *phase=0.0*, *offset=0.0*, *mult=1.0*)
Temp wrapper for new env function

dsp.**fth**(*frames*)
frames to hz

dsp.**htf**(*hz*)
hz to frames

dsp.**iscrossing**(*first*, *second*)
Detects zero crossing between two mono frames

dsp.**read**(*filename*)
Read a 44.1k / 16bit WAV file from disk with the Python wave module. Mono files are converted to stereo automatically.

dsp.**split**(*string*, *size*, *chans=2*)
split a sound into chunks of size N in frames, or by zero crossings

dsp.**splitmono**(*string*)
split a stereo sound into a list of mono sounds

dsp.**stretch**(*snd*, *length=None*, *speed=None*, *grain_size=120*)
Crude granular time stretching and pitch shifting

dsp.**tone**(*length=44100*, *freq=440.0*, *wavetype='sine'*, *amp=1.0*, *phase=0.0*, *offset=0.0*)
Synthesize a tone with the given params.

Possible wavetypes:

- sine2pi or sine
- cos2pi or cos
- hann
- tri
- saw or line
- isaw or phasor
- vary
- impulse
- square

Length is given in frames.

dsp.**transpose**(*snd*, *amount*)
Change the speed of a sound. 1.0 is unchanged, 0.5 is half speed, 2.0 is twice speed, etc.

This is a wrapper for audioop.ratecv in the standard library.

dsp.**wavetable**(*wtype='sine'*, *size=512*, *highval=1.0*, *lowval=0.0*)
The end is near. That'll do, wavetable()

dsp.**write**(*audio_string*, *filename*, *timestamp=False*, *cwd=True*)
Write audio data to renders directory with the Python wave module

lists.**eu**(*length*, *numpulses*)
A euclidian pattern generator

```
>>> dsp.eu(12, 3)
[1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]

>>> dsp.eu(12, 5)
[1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0]
```

lists.**interleave**(*list_one*, *list_two*)
Interleave the elements of two lists.

```
>>> dsp.interleave([1,1], [0,0])
[1, 0, 1, 0]
```

lists.**list_split**(*items*, *packet_size*)
Split a list into lists of a given size

```
>>> dsp.list_split(range(10), 3)
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

lists.**packet_shuffle**(*items*, *size*)
Shuffle the items in a list at a given granularity >= 3

```
>>> dsp.packet_shuffle(range(10), 3)
[0, 1, 2, 5, 4, 3, 8, 7, 6]

>>> dsp.packet_shuffle(range(10), 3)
[1, 2, 0, 4, 3, 5, 7, 6, 8]
```

lists.**rotate**(*items*, *start=0*, *vary=False*)
Rotate a list by a given (optionally variable) offset

```
>>> dsp.rotate(range(10), 3)
[3, 4, 5, 6, 7, 8, 9, 0, 1, 2]

>>> dsp.rotate(range(10), vary=True)
[6, 7, 8, 9, 0, 1, 2, 3, 4, 5]

>>> dsp.rotate(range(10), vary=True)
[8, 9, 0, 1, 2, 3, 4, 5, 6, 7]
```

utils.**byte_string**(*number*)
Takes an integer (truncated to between -32768 and 32767) and returns a single frame of sound.

utils.**cap**(*num*, *max*, *min=0*)
Takes a number and a maximum and minimum cap and returns a truncated number within (inclusive) that range:

```
>>> dsp.cap(500, 1, 0)
1

>>> dsp.cap(-42424242, 32767, -32768)
-32768
```

utils.**flen**(*snd*)
Returns the length of a sound in frames

```
>>> dsp.flen(dsp.tone(dsp.mstf(1))) == dsp.mstf(1)
True
```

utils.**log**(*message*, *mode='a'*)
Write to a temporary log file at ~/pippi.log for debugging. Set mode to "w" or similar to truncate logs when starting a new session.

utils.**pack**(*number*)
Takes a float between -1.0 to 1.0 and returns a single frame of sound. A wrapper for byte_string()

utils.**scale**(*low_target*, *high_target*, *low*, *high*, *pos*)
Takes a target range (low, high) and source range (low, high) and a value in the source range and returns a scaled value in the target range.

To scale a value between 0 and 1 to a value between 0 and 100:

```
>>> print dsp.scale(0, 100, 0, 1, 0.5)
50.0
```

utils.**timer**(*cmd='start'*)
Coarse time measurement useful for non-realtime scripts.

Start the timer at the top of your script:

```
dsp.timer('start')
```

And stop it at the end of your script to print the time elapsed in seconds. Make sure dsp.quiet is False so the elapsed time will print to the console.

```
dsp.timer('stop')
```

utils.**timestamp_filename**()
Convenience function that formats a datetime string for filenames:

```
>>> dsp.timestamp_filename()
```

'2015-10-21_07.28.00'

tune.**ntf**(*note*, *octave=4*, *ratios=None*)
Note to freq

tune.**nti**(*note*)
Note to index returns the index of enharmonic note names or False if not found

_pippic.**add**()
Add two sounds together.

_pippic.**am**()
Multiply two sounds together.

_pippic.**amp**()
Multiply a sound by a constant.

_pippic.**curve**()
Envelope and control curve generation.

_pippic.**cycle**()
Generate a single cycle of a waveform.

_pippic.**env**()
Apply an envelope to a sound.

_pippic.**fold**()
Wave folding synthesis.

_pippic.**invert**()
Invert a sound.

_pippic.**mix**()
Mix an arbitrary number of sounds together.

_pippic.**mul**()
Multiply two sounds together.

_pippic.**pine**()
Just your average pinecone.

_pippic.**pulsar**()
Pulsar synthesis.

_pippic.**shift**()
Change speed.

_pippic.**subtract**()
Subtract one sound from another.

_pippic.**synth**()
Synthy. Depreciated: use tone() with a python wavetable instead

_pippic.**tone**()
Wavetable synthesis.

_pippic.**wtread**()
Read from a wavetable

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Symbols

A

B

C

D

E

F

H

I

L

M

N

P

R

S

T

U

W