# Realtime Audio Visualization in Python

July 19, 2016 python obsolete

⚠️ **WARNING: This page is obsolete**
Articles typically receive this designation when the technology they describe is no longer relevant, code provided is later deemed to be of poor quality, or the topics discussed are better presented in future articles. Articles like this are retained for the sake of preservation, but their content should be critically assessed.

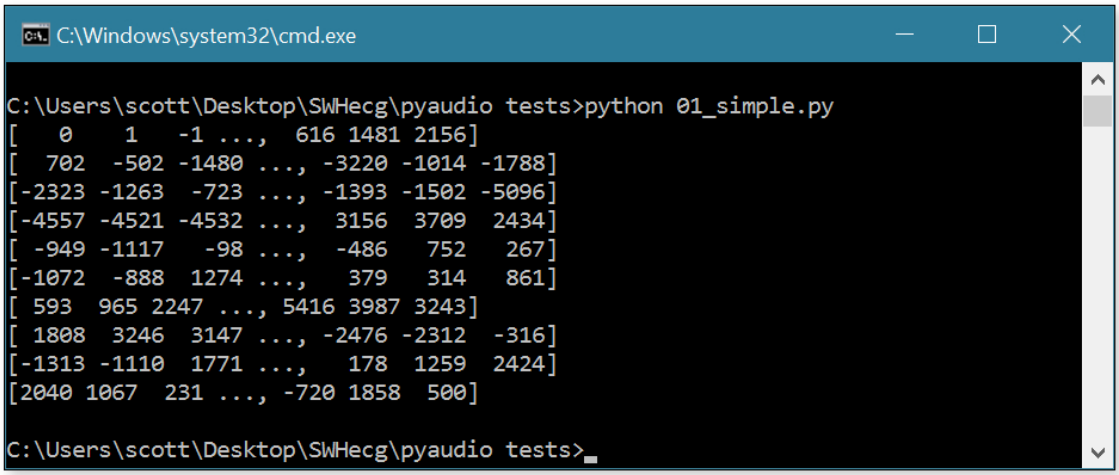
**Python's "batteries included" nature makes it easy to interact with just about anything... except speakers and a microphone!** As of this moment, there still are not standard libraries which which allow cross-platform interfacing with audio devices. There are some pretty convenient third-party modules, but I hope in the future a standard solution will be distributed with python. I appreciate the differences of Linux architectures such as ALSA and OSS, but toss in Windows and MacOS in the mix and it gets to be a huge mess. For Linux, would I even need anything fancy? I can run "`cat file.wav > /dev/dsp`" from a command prompt to play audio. There are some standard libraries for operating system specific sound (i.e., winsound), but I want something more versatile. The official audio wiki page on the subject lists a small collection of third-party platform-independent libraries. After excluding those which don't support microphone access (the ultimate goal of all my poking around in this subject), I dove a little deeper into sounddevice and PyAudio. Both of these I installed with pip (i.e., `pip install pyaudio`)

**I really like the structure and documentation of sounddevice, but I decided to keep developing with PyAudio for now.** Sounddevice seemed to take more system resources than PyAudio (in my limited test conditions: Windows 10 with very fast and modern hardware, Python 3), and would audibly "glitch" music as it was being played every time it attached or detached from the microphone stream. I tried streaming, but after about an hour I couldn't get clean live access to the microphone without glitching audio playback. Furthermore, every few times I ran this script it crashed my python kernel! I very rarely see this happening. iPython complained: *"It seems the kernel died unexpectedly. Use 'Restart kernel' to continue using this console"* and I eventually moved back to PyAudio. For a less "realtime" application, sounddevice might be a great solution. Here's the minimal case sounddevice script I tested with (that crashed sometimes). If you have a better one to do live high-speed audio capture, let me know!

```
import sounddevice #pip install sounddevice

for i in range(30): #30 updates in 1 second
    rec = sounddevice.rec(44100/30)
    sounddevice.wait()
    print(rec.shape)
```

**Here's a simple demo to show how I get realtime microphone audio into numpy arrays using PyAudio.** This isn't really that special. It's a good starting point though. Note that rather than have the user define a microphone source in the python script (I had a fancy menu system handling this for a while), I allow PyAudio to just look at the operating system's default input device. This seems like a realistic expectation, and saves time as long as you don't expect your user to be recording from two different devices at the same time. This script gets some audio from the microphone and shows the values in the console (ten times).

```
import pyaudio
import numpy as np

CHUNK = 4096 # number of data points to read at a time
RATE = 44100 # time resolution of the recording device (Hz)

p=pyaudio.PyAudio() # start the PyAudio class
stream=p.open(format=pyaudio.paInt16,channels=1,rate=RATE,input=True,
              frames_per_buffer=CHUNK) #uses default input device

# create a numpy array holding a single read of audio data
for i in range(10): #to it a few times just to see
    data = np.fromstring(stream.read(CHUNK),dtype=np.int16)
    print(data)

# close the stream gracefully
stream.stop_stream()
stream.close()
p.terminate()
```

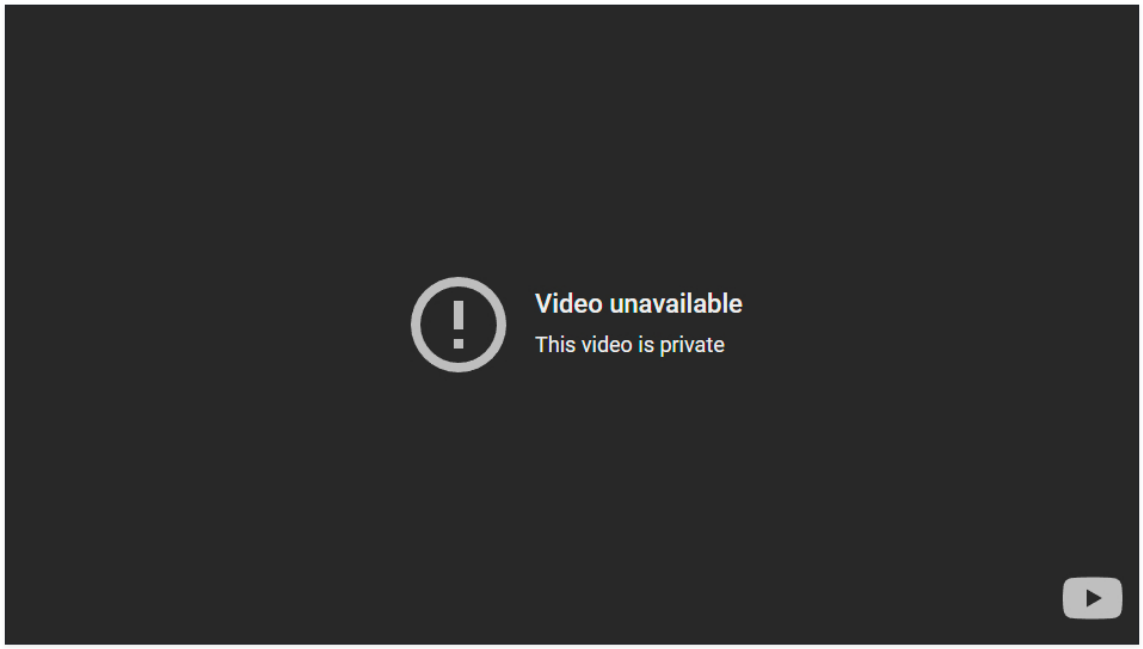



I tried to push the limit a little bit and see how much useful data I could get from this console window. It turns out that it's pretty responsive! Here's a slight modification of the code, made to turn the console window into an impromptu VU meter.

```
import pyaudio
import numpy as np

CHUNK = 2**11
RATE = 44100

p=pyaudio.PyAudio()
stream=p.open(format=pyaudio.paInt16,channels=1,rate=RATE,input=True,
              frames_per_buffer=CHUNK)

for i in range(int(10*44100/1024)): #go for a few seconds
    data = np.fromstring(stream.read(CHUNK),dtype=np.int16)
    peak=np.average(np.abs(data))*2
    bars="#"*int(50*peak/2**16)
    print("%04d %05d %s"%(i,peak,bars))

stream.stop_stream()
stream.close()
p.terminate()
```

## Result

The results are pretty good! The advantage here is that *no* libraries are required except PyAudio. For people interested in doing simple math (peak detection, frequency detection, etc.) this is a perfect starting point. Here's a quick cellphone video:



I've made realtime audio visualization (realtime FFT) scripts with Python before, but 80% of that code was creating a GUI. I want to see data in real time while I'm developing this code, but I *really* don't want to mess with GUI programming. I then had a crazy idea. Everyone has a web browser, which is a pretty good GUI... with a Python script to analyze audio and save graphs (a lot of them, quickly) and some JavaScript running in a browser to keep refreshing those graphs, I could get an idea of what the audio stream is doing in something kind of like real time. It was intended to be a hack, but I never expected it to work so well! Check this out...

**Here's the python script to listen to the microphone and generate graphs:**

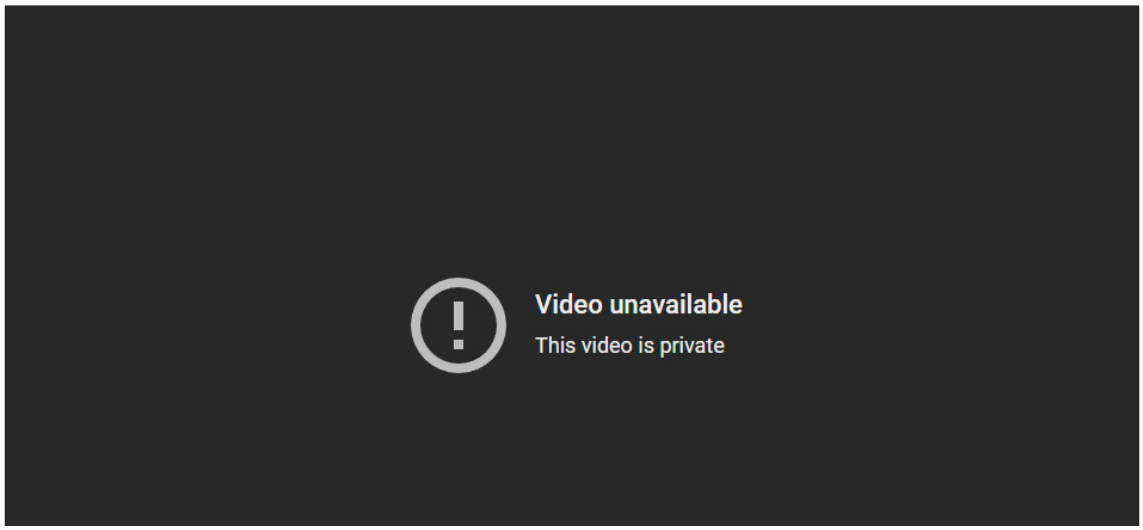
```
import pyaudio
import numpy as np
import pylab
import time

RATE = 44100
CHUNK = int(RATE/20) # RATE / number of updates per second

def soundplot(stream):
    t1=time.time()
    data = np.fromstring(stream.read(CHUNK),dtype=np.int16)
    pylab.plot(data)
    pylab.title(i)
    pylab.grid()
    pylab.axis([0,len(data),-2**16/2,2**16/2])
    pylab.savefig("03.png",dpi=50)
    pylab.close('all')
    print("took %.02f ms"%((time.time()-t1)*1000))

if __name__=="__main__":
    p=pyaudio.PyAudio()
    stream=p.open(format=pyaudio.paInt16,channels=1,rate=RATE,input=True,
                  frames_per_buffer=CHUNK)
    for i in range(int(20*RATE/CHUNK)): #do this for 10 seconds
        soundplot(stream)
    stream.stop_stream()
    stream.close()
    p.terminate()
```

**Here's the HTML file with JavaScript to keep reloading the image...**

```
<html>
<script language="javascript">
function RefreshImage(){
document.pic0.src="https://swharden.com/static/2016/07/19/03.png?a=" + String(Math.random()*99999999);
setTimeout('RefreshImage()',50);
}
</script>
<body onload="RefreshImage()">
<img name="pic0" src="https://swharden.com/static/2016/07/19/03.png">
</body>
</html>
```

## Operation

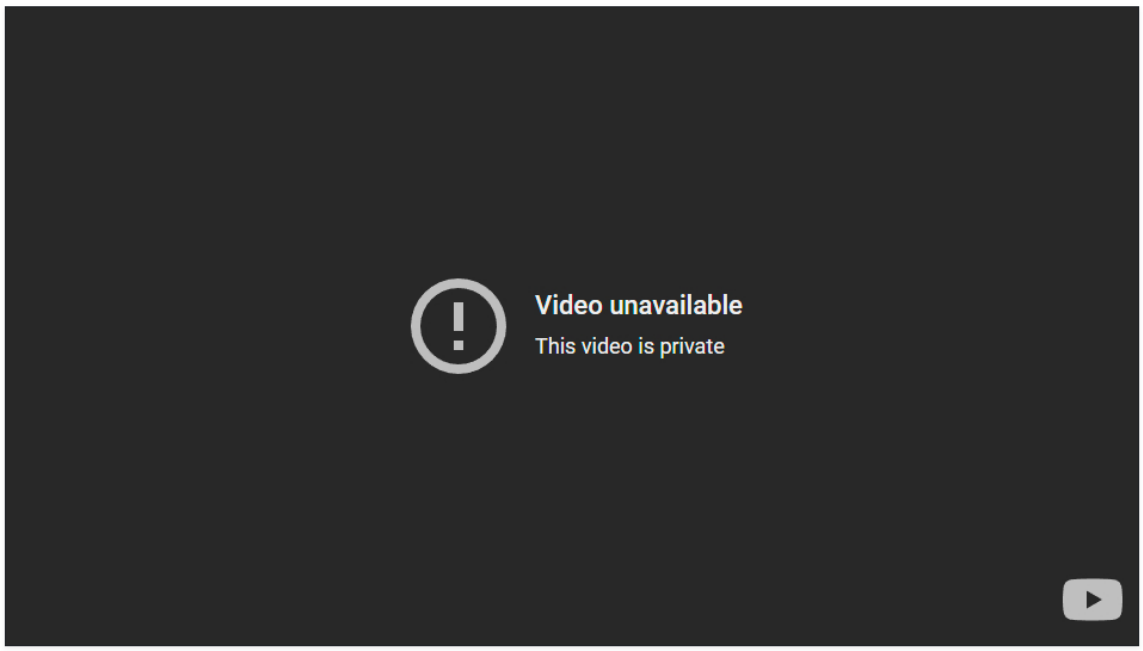
I couldn't believe my eyes. It's not elegant, but it's kind of functional!



**Why stop there?** I went ahead and wrote a microphone listening and processing class which makes this stuff easier. My ultimate goal hasn't been revealed yet, but I'm sure it'll be clear in a few weeks. Let's just say there's a lot of use in me visualizing streams of continuous data. Anyway, this class is the truly *terrible* attempt at a word pun by merging the words "SWH", "ear", and "Hear", into the official title "SWHear" which seems to be unique on Google. This class is minimal case, but can be easily modified to implement threaded recording (which won't cause the rest of the functions to hang) as well as mathematical manipulation of data, such as FFT. With the same HTML file as used above, here's the new python script and some video of the output:

```
import pyaudio
import time
import pylab
import numpy as np

class SWHear(object):
    """
    The SWHear class is made to provide access to continuously recorded
    (and mathematically processed) microphone data.
    """

    def __init__(self,device=None,startStreaming=True):
        """fire up the SWHear class."""
        print(" -- initializing SWHear")

        self.chunk = 4096 # number of data points to read at a time
        self.rate = 44100 # time resolution of the recording device (Hz)

        # for tape recording (continuous "tape" of recent audio)
        self.tapeLength=2 #seconds
        self.tape=np.empty(self.rate*self.tapeLength)*np.nan

        self.p=pyaudio.PyAudio() # start the PyAudio class
        if startStreaming:
            self.stream_start()

    ### LOWEST LEVEL AUDIO ACCESS
    # pure access to microphone and stream operations
    # keep math, plotting, FFT, etc out of here.

    def stream_read(self):
        """return values for a single chunk"""
        data = np.fromstring(self.stream.read(self.chunk),dtype=np.int16)
        #print(data)
        return data

    def stream_start(self):
        """connect to the audio device and start a stream"""
        print(" -- stream started")
        self.stream=self.p.open(format=pyaudio.paInt16,channels=1,
                                rate=self.rate,input=True,
                                frames_per_buffer=self.chunk)

    def stream_stop(self):
        """close the stream but keep the PyAudio instance alive."""
        if 'stream' in locals():
            self.stream.stop_stream()
            self.stream.close()
        print(" -- stream CLOSED")

    def close(self):
        """gently detach from things."""
        self.stream_stop()
        self.p.terminate()

    ### TAPE METHODS
    # tape is like a circular magnetic ribbon of tape that's continously
    # recorded and recorded over in a loop. self.tape contains this data.
    # the newest data is always at the end. Don't modify data on the type,
    # but rather do math on it (like FFT) as you read from it.

    def tape_add(self):
        """add a single chunk to the tape."""
        self.tape[:-self.chunk]=self.tape[self.chunk:]
        self.tape[-self.chunk:]=self.stream_read()
```

```
    def tape_flush(self):
        """completely fill tape with new data."""
        readsInTape=int(self.rate*self.tapeLength/self.chunk)
        print(" -- flushing %d s tape with %dx%.2f ms reads"%\
                  (self.tapeLength,readsInTape,self.chunk/self.rate))
        for i in range(readsInTape):
            self.tape_add()

    def tape_forever(self,plotSec=.25):
        t1=0
        try:
            while True:
                self.tape_add()
                if (time.time()-t1)>plotSec:
                    t1=time.time()
                    self.tape_plot()
        except:
            print(" ~~ exception (keyboard?)")
            return

    def tape_plot(self,saveAs="03.png"):
        """plot what's in the tape."""
        pylab.plot(np.arange(len(self.tape))/self.rate,self.tape)
        pylab.axis([0,self.tapeLength,-2**16/2,2**16/2])
        if saveAs:
            t1=time.time()
            pylab.savefig(saveAs,dpi=50)
            print("plotting saving took %.02f ms"%((time.time()-t1)*1000))
        else:
            pylab.show()
            print() #good for IPython
        pylab.close('all')

if __name__=="__main__":
    ear=SWHear()
    ear.tape_forever()
    ear.close()
    print("DONE")
```




**I don't really intend anyone to actually do this,** but it's a cool alternative to recording a small portion of audio, plotting it in a pop-up matplotlib window, and waiting for the user to close it to record a new fraction. I had a lot more text in here demonstrating real-time FFT, but I'd rather consolidate everything FFT related into a single post. For now, I'm happy pursuing microphone-related python projects with PyAudio.

## Display a single frequency

Use Numpy's FFT() and FFTFREQ() to turn the linear data into frequency. Set that target and grab the FFT value corresponding to that frequency. I haven't tested this to be sure it's working, but it should at least be close

frequency. I haven't tested this to be sure it's working, but it should at least be close...

```
import pyaudio
import numpy as np
np.set_printoptions(suppress=True) # don't use scientific notation

CHUNK = 4096 # number of data points to read at a time
RATE = 44100 # time resolution of the recording device (Hz)
TARGET = 2100 # show only this one frequency

p=pyaudio.PyAudio() # start the PyAudio class
stream=p.open(format=pyaudio.paInt16,channels=1,rate=RATE,input=True,
              frames_per_buffer=CHUNK) #uses default input device

# create a numpy array holding a single read of audio data
for i in range(10): #to it a few times just to see
    data = np.fromstring(stream.read(CHUNK),dtype=np.int16)
    fft = abs(np.fft.fft(data).real)
    fft = fft[:int(len(fft)/2)] # keep only first half
    freq = np.fft.fftfreq(CHUNK,1.0/RATE)
    freq = freq[:int(len(freq)/2)] # keep only first half
    assert freq[-1]>TARGET, "ERROR: increase chunk size"
    val = fft[np.where(freq>TARGET)[0][0]]
    print(val)

# close the stream gracefully
stream.stop_stream()
stream.close()
p.terminate()
```

## Display Peak Frequency

If your goal is to determine which frequency is producing the loudest tone, use this function. I also added a few lines to graph the output in case you want to observe how it operates. I recommend testing this script with a tone generator, or a YouTube video containing tones of a range of frequencies like this one.

```
import pyaudio
import numpy as np
import matplotlib.pyplot as plt

np.set_printoptions(suppress=True) # don't use scientific notation

CHUNK = 4096 # number of data points to read at a time
RATE = 44100 # time resolution of the recording device (Hz)

p=pyaudio.PyAudio() # start the PyAudio class
stream=p.open(format=pyaudio.paInt16,channels=1,rate=RATE,input=True,
              frames_per_buffer=CHUNK) #uses default input device

# create a numpy array holding a single read of audio data
for i in range(10): #to it a few times just to see
    data = np.fromstring(stream.read(CHUNK),dtype=np.int16)
    data = data * np.hanning(len(data)) # smooth the FFT by windowing data
    fft = abs(np.fft.fft(data).real)
    fft = fft[:int(len(fft)/2)] # keep only first half
    freq = np.fft.fftfreq(CHUNK,1.0/RATE)
    freq = freq[:int(len(freq)/2)] # keep only first half
    freqPeak = freq[np.where(fft==np.max(fft))[0][0]]+1
    print("peak frequency: %d Hz"%freqPeak)

    # uncomment this if you want to see what the freq vs FFT looks like
    #plt.plot(freq,fft)
    #plt.axis([0,4000,None,None])
    #plt.show()
    #plt.close()

# close the stream gracefully
stream.stop_stream()
stream.close()
p.terminate()
```

## Display Left and Right Levels

```
import pyaudio
import numpy as np

maxValue = 2**16
```

```
p=pyaudio.PyAudio()
stream=p.open(format=pyaudio.paInt16,channels=2,rate=44100,
              input=True, frames_per_buffer=1024)
while True:
    data = np.fromstring(stream.read(1024),dtype=np.int16)
    dataL = data[0::2]
    dataR = data[1::2]
    peakL = np.abs(np.max(dataL)-np.min(dataL))/maxValue
    peakR = np.abs(np.max(dataR)-np.min(dataR))/maxValue
    print("L:%00.02f R:%00.02f"%(peakL*100, peakR*100))
```

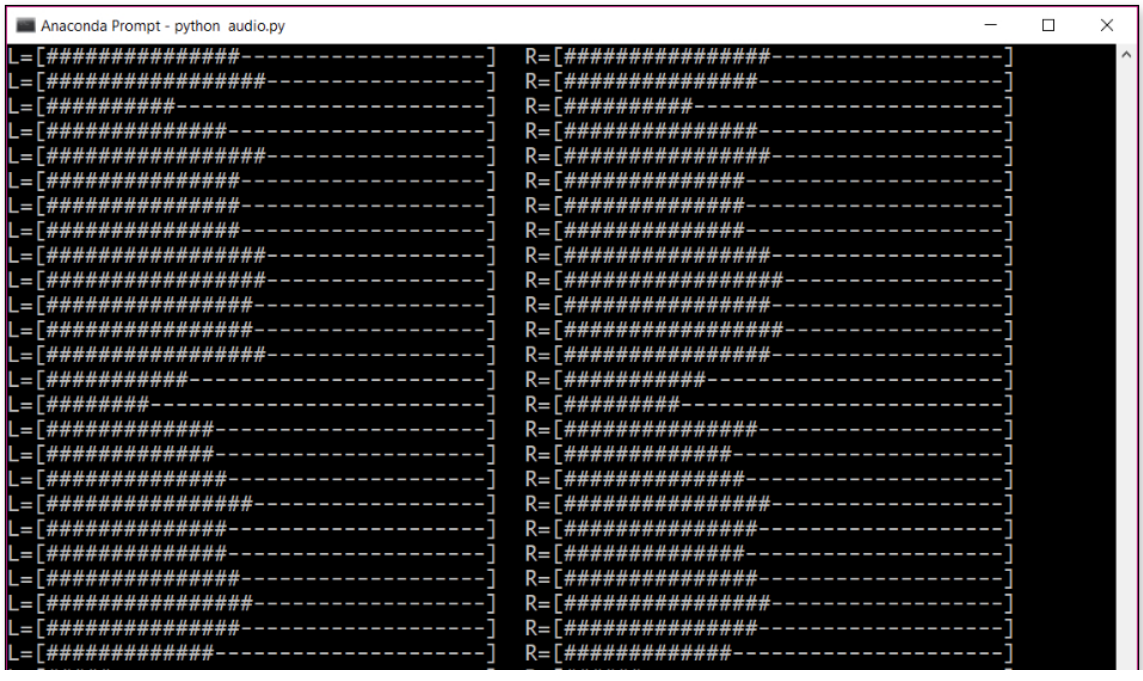
**Output**

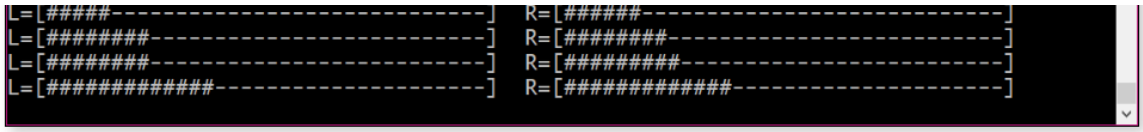
```
L:47.26 R:45.17
L:47.55 R:45.63
L:49.44 R:45.98
L:45.27 R:49.80
L:44.39 R:45.75
L:47.50 R:46.96
L:41.49 R:42.64
L:42.95 R:41.39
L:49.56 R:49.62
L:48.29 R:48.80
L:45.03 R:47.62
L:47.99 R:49.35
L:41.58 R:49.21
```

Or with a tweak...

```
import pyaudio
import numpy as np

maxValue = 2**16
bars = 35
p=pyaudio.PyAudio()
stream=p.open(format=pyaudio.paInt16,channels=2,rate=44100,
              input=True, frames_per_buffer=1024)
while True:
    data = np.fromstring(stream.read(1024),dtype=np.int16)
    dataL = data[0::2]
    dataR = data[1::2]
    peakL = np.abs(np.max(dataL)-np.min(dataL))/maxValue
    peakR = np.abs(np.max(dataR)-np.min(dataR))/maxValue
    lString = "#"*int(peakL*bars)+"-"*int(bars-peakL*bars)
    rString = "#"*int(peakR*bars)+"-"*int(bars-peakR*bars)
    print("L=[%s]\tR=[%s]"%(lString, rString))
```

## Graphical Output






Feedback Source Contact


## Newer

Opto-Isolated Laser Controller Build

July 28, 2016

## Older

Controlling Bus Pirate with Python

July 14, 2016