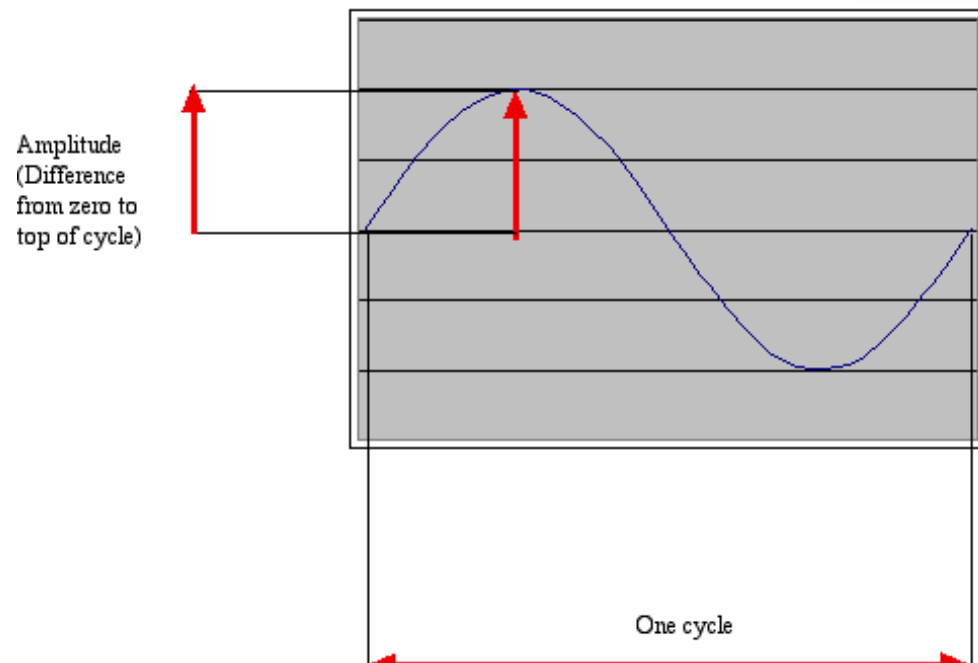


# **Tinkering Audio II: Further Notes on Digital Sound**

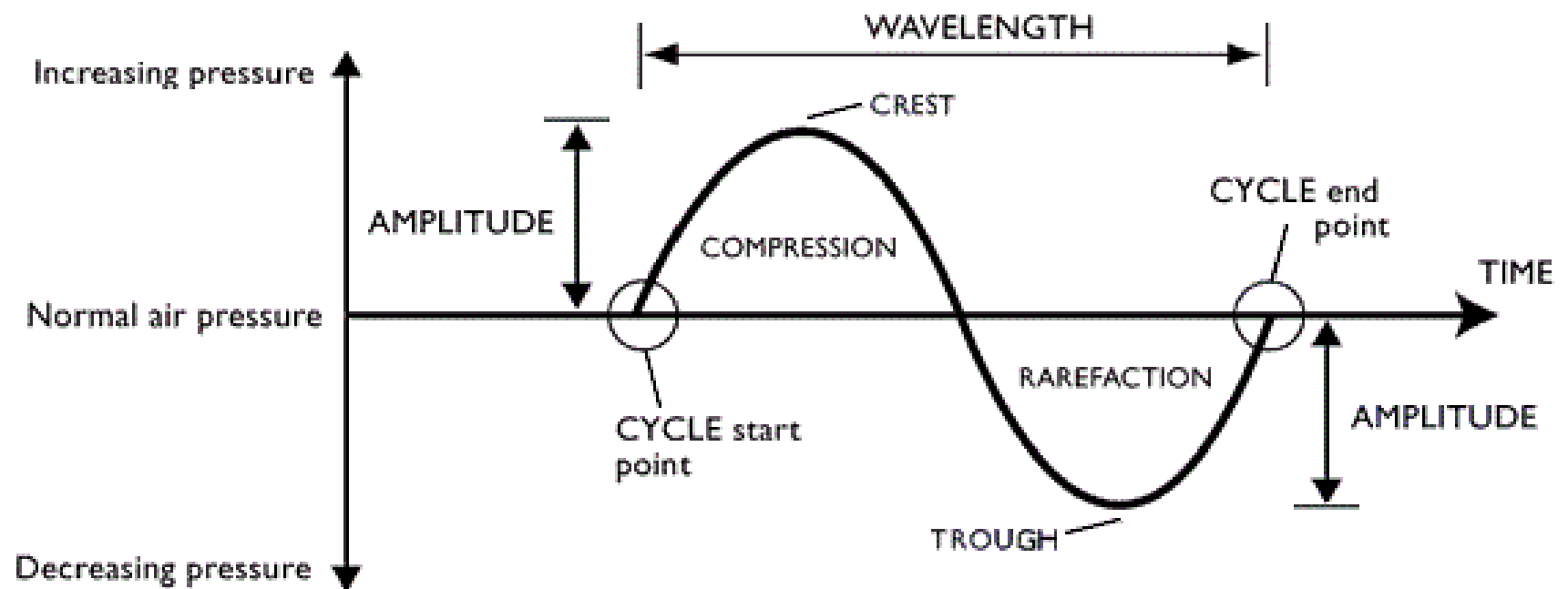
Creative Computing: Tinkering – Lecture 9 – Michael Scott

# Recap on Last Week

- Sounds are waves of air pressure
  - Sound comes in cycles
  - The *frequency* of a wave is the number of cycles per second (cps), or *Hertz*
  - Complex sounds have more than one frequency in them.



# Recap on Last Week



# Recap on Last Week

- Our perception of volume is related (logarithmically) to changes in amplitude
  - If the amplitude doubles, it's about a 3 decibel (dB) change
- Our perception of pitch is related (logarithmically) to changes in frequency
  - Higher frequencies are perceived as higher pitches
  - We can hear between 5 Hz and 20,000 Hz (20 kHz)
  - A above middle C is 440 Hz



# Example: Shepard Tones

<http://www.moillusions.com/wp-content/uploads/2006/05/shepards.mp3>



# “Logarithmically?” – Further Notes

- It's strange, but our perception relies on ratios rather than absolute differences, e.g., for pitch.
  - We hear changes between 200 Hz and 400 Hz, as the same as changes between 500 Hz and 1000 Hz (1:2)
  - Similarly, 200 Hz to 600 Hz, and 1000 Hz to 3000 Hz, etc. (1:3)
  - Repeatedly multiplying amplitude by a constant factor is perceived by humans as a series of equal increases.
- Intensity (volume) is measured as watts per meter squared
  - A change from  $0.1\text{W}/\text{m}^2$  to  $0.01\text{W}/\text{m}^2$ , sounds the same to us as  $0.001\text{W}/\text{m}^2$  to  $0.0001\text{W}/\text{m}^2$



# “Logarithmically?” – Further Notes

- A decibel is a ratio between two intensities:

$$10 * \log_{10}(I_1/I_2)$$

- As an absolute measure, it's in comparison to threshold of audibility
- 0 dB can't be heard.
- Normal speech is 60 dB.
- A shout is about 80 dB

Tinkering Audio

# TONE GENERATION





# Tone Generator

## Example

```
noise_out = wave.open(FILE, 'w')
noise_out.set_basic_params(
    noise_out.get_params() + (
        SAMPLE_RATE, CHANNELS, 2, 16))

values = []

for i in range(0, SAMPLE_LENGTH):
    value = sin(
        2.0 * PI *
        FREQUENCY *
        (i / SAMPLE_RATE)
    ) *
    (VOLUME * BIT_DEPTH)

    packaged_value = package(FORMAT, value)

    for j in xrange(0, CHANNELS):
        values.append(packaged_value)

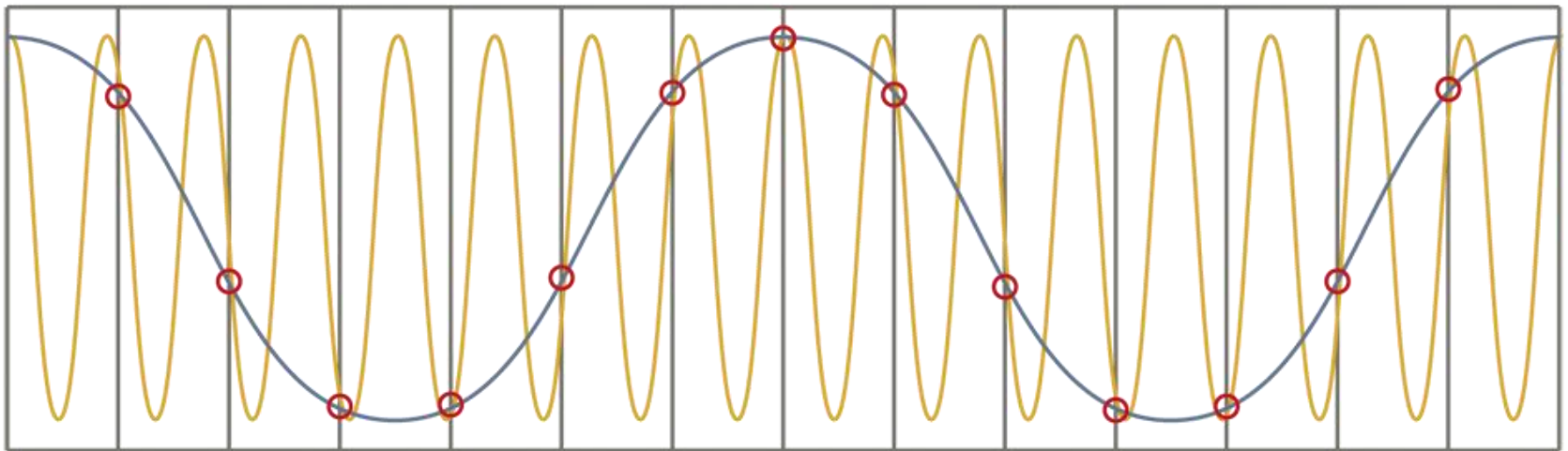
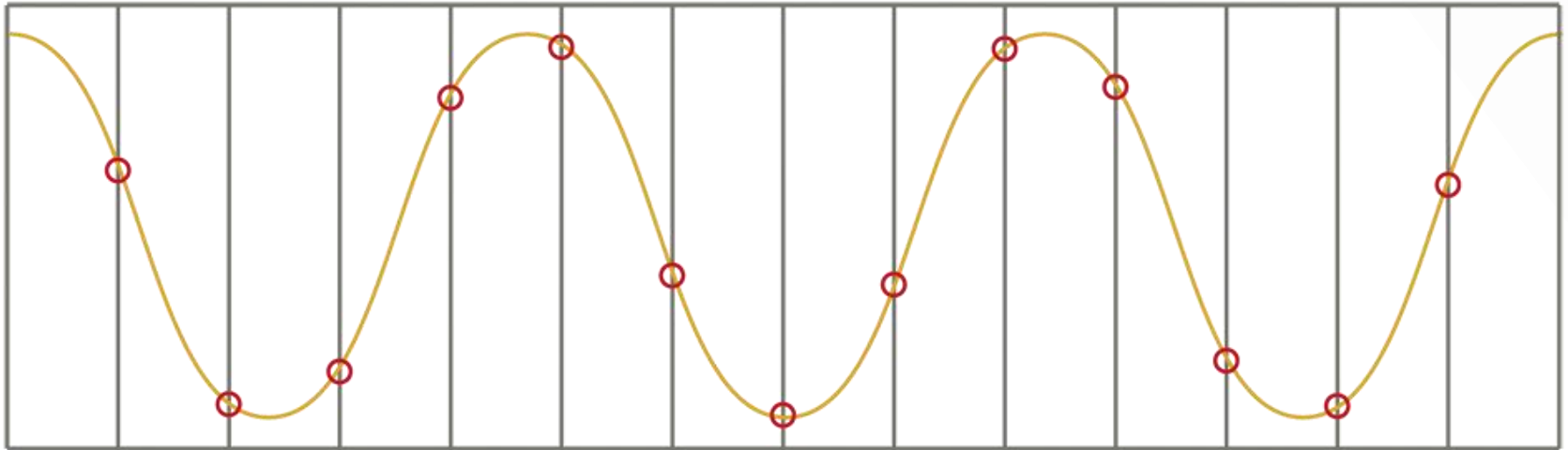
value_str = ''.join(values)
noise_out.write(value_str)

noise_out.close()
```

## Explanation

- What should the parameters be?
- Why this formula?  
Why  $2 * \pi$ ?  
Why  $VOLUME * BIT\_DEPTH$ ?  
Why  $SAMPLE\_RATE$ ?
- Why append extra values?

# Aliasing



# Nyquist Theorem

- We need twice as many samples as the maximum frequency in order to represent (and recreate, later) the original sound.
- The number of samples recorded per second is the sampling rate
  - If we capture 8000 samples per second, the highest frequency we can capture is 4000 Hz
    - That's how phones work
  - If we capture more than 44,000 samples per second, we capture everything that we can hear (max 22,000 Hz)
    - CD quality is 44,100 samples per second



# Tones and Tone Generation

- Has anyone generated tones using  $\sin()$  waves?
  - Post a simple solution on Slack
- What is the maximum possible amplitude? and Why?
  - Discuss on Slack for 5 minutes
- What happens if you keep increasing the frequency?
  - Discuss on Slack for 5 minutes





Tinkering Audio

# TONE COMBINATION

# Fourier and Superposition

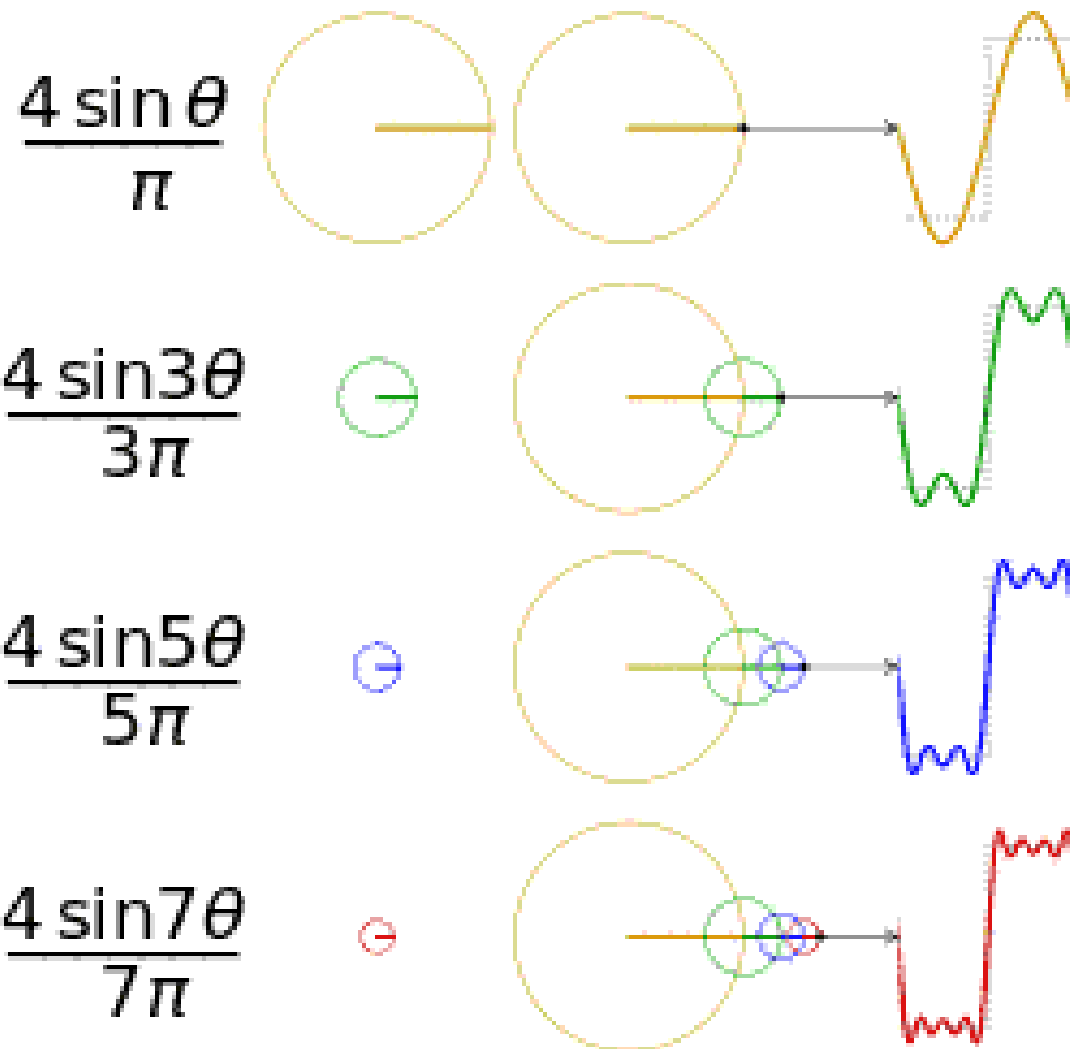
- Prior to Fourier's work, no solution to a particular complex equation was known in the general case, although particular solutions were known if the variables behaved in a simple way; in particular, following the sine or cosine functions.
- These simple solutions are now sometimes called "eigensolutions".



# Fourier and Superposition

- Fourier's idea was to model a complicated variable as a superposition (or linear combination) of simple sine and cosine waves, and then to write the solution as a superposition of the corresponding eigensolutions.
- This superposition (or linear combination) is now called the Fourier series. Correspondingly, Fourier series' can be used in the construction of complex tones.

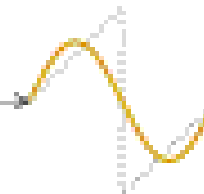
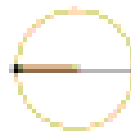
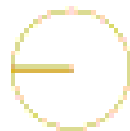




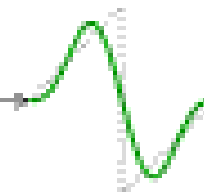
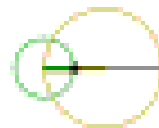
**Approximating a Square Wave**



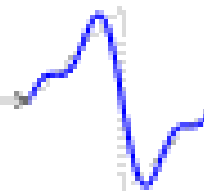
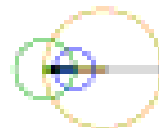
$$\frac{2\sin\theta}{-\pi}$$



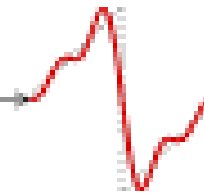
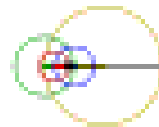
$$\frac{2\sin 2\theta}{2\pi}$$



$$\frac{2\sin 3\theta}{-3\pi}$$



$$\frac{2\sin 4\theta}{4\pi}$$



Approximating a Saw-Tooth Wave



<https://www.youtube.com/watch?v=YsZKvLnf7wU>

# Writing a Tone Combination Function

- Write a function to combine two tones from your generator together.
- Once you have done this, attempt to re-create a saw-tooth and/or square wave.





Tinkering Audio

# NORMALISATION AND CLIPPING

# Learning Objectives

By the end of this section, you will be able to:

- **Explain** the terms normalization and clipping
- **Write** a function that will normalise a sound



# Normalisation

"the application of a constant amount of gain to an audio recording to bring the average or peak amplitude to a target level (the norm)"



# Normalizing Sound

```
def normalize(sound):  
    largest = 0  
    for s in getSamples(sound):  
        largest = max(largest, getSampleValue(s))  
    amplification = 32767.0 / largest
```

This loop finds the loudest sample

```
print "Largest sample value in original sound was", largest  
print "Amplification multiplier is", amplification
```

Why  
32,767.0?

```
for s in getSamples(sound):  
    louder = amplification * getSampleValue(s)  
    setSampleValue(s, louder)
```

This loop actually amplifies the sound

# Avoiding clipping

- Why are we being so careful to stay within range? What if we just multiplied all the samples by some big number and let some of them go over 32,767?
- The result then is *clipping*
  - Clipping: The awful, buzzing noise whenever the sound volume is beyond the maximum that your sound system can handle.

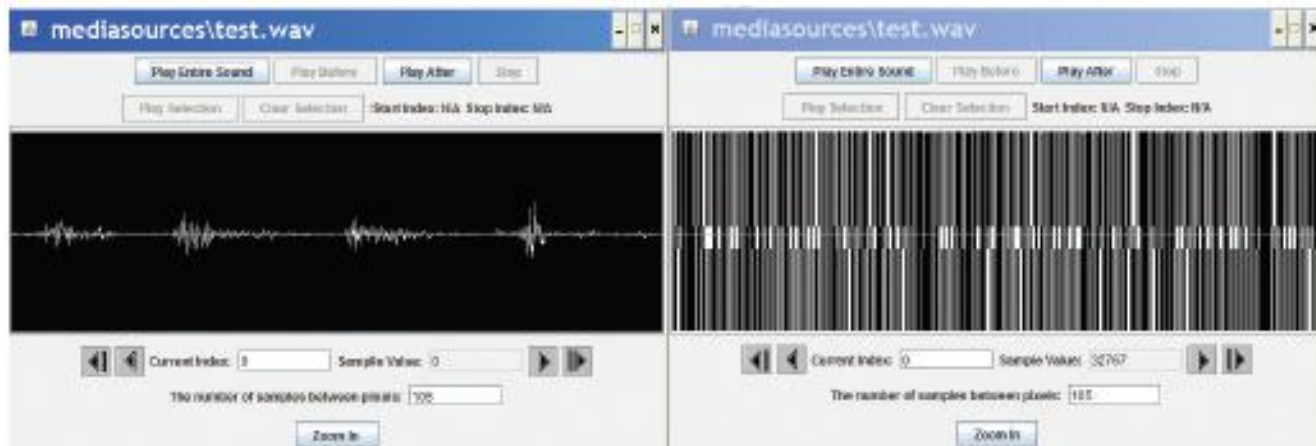


# All Clipping, All the Time

```
def maximize(sound):  
    for sample in getSamples(sound):  
        value = getSampleValue(sample)  
        if value > 0:  
            setSampleValue(sample, 32767)  
        if value < 0:  
            setSampleValue(sample, -32768)
```

What happens when we maximise a sound?

- All samples over 0: Make it 32767
- All samples at or below 0: Make it -32768



# We can hear the speech!

- Try it! You can understand speech in this mangled sound.
- Why?
- Implications:
  - Human understanding of speech relies more on *frequency* than *amplitude*.
  - Note how many *bits* we need per sample. A single bit per sample can record (somewhat) legible speech.





Introduction to Digital Sound

# **SPLICING AND SWAPPING AUDIO**

# Processing only part of the sound

- What if we wanted to increase or decrease the volume of only part of the sound?
- Q: How would we do it?
- A: We'd have to use a `range()` function with our for loop
  - Just like when we manipulated only part of a picture by using `range()` in conjunction with `getPixels()`



# Using for to count with range

```
>>> print range(1,3)
[1, 2]
>>> print range(3,1)
[]
>>> print range(-1,5)
[-1, 0, 1, 2, 3, 4]
>>> print range(1,100)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... 99]
```

# Knowing where we are in the sound

- More complex operations require us to know where we are in the sound, which sample
  - Not just process all the samples exactly the same
- Examples:
  - **Reversing** a sound
    - It's just copying, like we did with pixels
  - **Changing the frequency** of a sound
    - Using sampling, like we did with pixels
  - **Splicing** sounds



# Increasing volume by *sample index*

```
def increaseVolumeByRange(sound):  
    for sampleNumber in range(0, getLength(sound)):  
        value = getSampleValueAt(sound, sampleNumber)  
        setSampleValueAt(sound, sampleNumber, value * 2)
```

```
def increaseVolume(sound):  
    for sample in getSamples(sound):  
        value = getSample(sample)  
        setSample(sample, value * 2)
```

What is the difference between these two functions?

# Modify different sound sections

The index lets us modify parts of the sound now - e.g. here we increase the volume in the first half, and then decrease it in the second half.

```
def increaseAndDecrease(sound):  
    length = getLength(sound)  
    for index in range(0, length/2):  
        value = getSampleValueAt(sound, index)  
        setSampleValueAt(sound, index, value*2)  
    for sampleIndex in range(length/2, length):  
        value = getSampleValueAt(sound, index)  
        setSampleValueAt(sound, index, value*0.2)
```



# Array References

Square brackets ([ ]) are standard notation for arrays (or lists). To access a single array element at position index, we use `array[index]`

```
>>> myArray = range(0, 100)
>>> print myArray[0]
0
>>> print myArray[1]
1
>>> print myArray[99]
99
```

# Splicing Sounds

- Splicing gets its name from literally cutting and pasting pieces of magnetic tape together
- Doing it digitally is easy (in principle), but painstaking
- The easiest kind of splicing is when the component sounds are in separate files.
- All we need to do is copy each sound, in order, into a target sound.
- Here's a recipe that creates the start of a sentence, “Guzdial is ...” (You may complete the sentence.)



# Splicing whole sound files

```
def merge():
    guzdial = makeSound(getMediaPath("guzdial.wav"))
    isSound = makeSound(getMediaPath("is.wav"))
    target = makeSound(getMediaPath("sec3silence.wav"))
    index = 0
    for source in range(0, getLength(guzdial)):
        value = getSampleValueAt(guzdial, source)
        setSampleValueAt(target, index, value)
        index = index + 1
    for source in range(0, int(0.1*getSamplingRate(target))):
        setSampleValueAt(target, index, 0)
        index = index + 1
    for source in range(0, getLength(isSound)):
        value = getSampleValueAt(isSound, source)
        setSampleValueAt(target, index, value)
        index = index + 1
    normalize(target)
    play(target)
    return target
```

# What additional functions must be in the file for that program to work?

1. `normalize()`
2. `play()`
3. `getMediaPath()`
4. `maximize()`

# How it works

- Creates sound objects for the words “Guzdial”, “is” and the target silence
- Set target's index to 0, then let each loop increment index and end the loop by leaving index at the next empty sample ready for the next loop
- The 1<sup>st</sup> loop copies “Guzdial” into the target
- The 2<sup>nd</sup> loop creates 0.1 seconds of silence
- The 3<sup>rd</sup> loop copies “is” into the target
- Then we normalize the sound to make it louder

# Splicing words into a speech

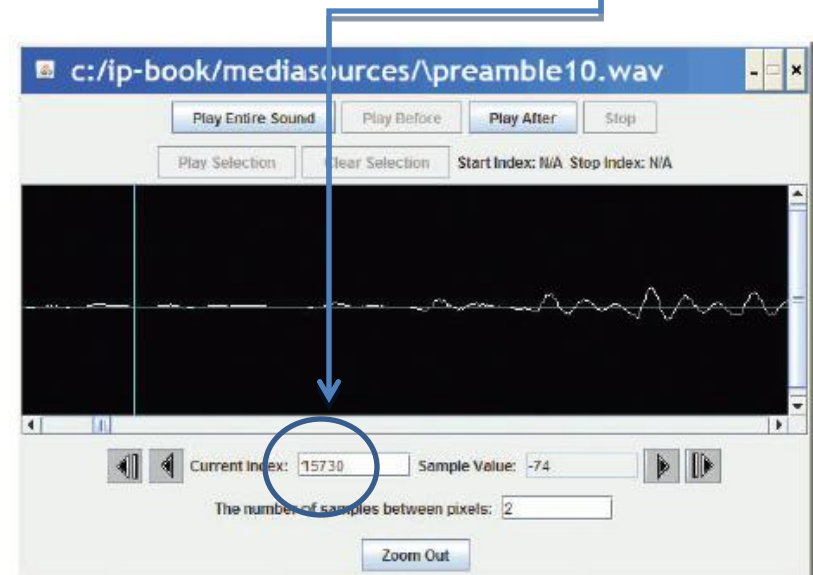
- Say we want to splice pieces of speech together:
  - We find where the end points of words are
  - We copy the samples into the right places to make the words come out as we want them
  - (We can also change the volume of the words as we move them, to increase or decrease emphasis and make it sound more natural.)



# Finding the word end-points

- Using MediaTools and play before/after cursor, we can figure out the index numbers where each word ends
- We want to splice a copy of the word “United” after “We the” so that it says, “We the United People of the United Kingdom”.

Word	Ending index
We	15730
the	17407
People	26726
of	32131
the	33413
United	40052
States	55510



# Now, it's all about copying

- We have to keep track of the source and target indices, **srcSample** and **destSample**

```
destSample = Where-the-incoming-sound-should-start
for srcSample in range(startingPoint, endingPoint):
    sampleValue = getSampleValueAt(source, srcSample)
    setSampleValueAt(dest, destSample, sampleValue)
    destSample = destSample + 1
```



# The Whole Splice

```
def splicePreamble():  
    file = getMediaPath("preamble10.wav")  
    source = makeSound(file)  
    target = makeSound(file) # This will be the newly spliced sound  
    targetIndex = 17408 # targetIndex starts at just after "We the" in the new sound  
    for sourceIndex in range(33414, 40052): # Where the word "United" is in the sound  
        setSampleValueAt(target, targetIndex, getSampleValueAt(source, sourceIndex))  
        targetIndex = targetIndex + 1  
    for sourceIndex in range(17408, 26726): # Where the word "People" is in the sound  
        setSampleValueAt(target, targetIndex, getSampleValueAt(source, sourceIndex))  
        targetIndex = targetIndex + 1  
    for index in range(0, 1000): # Stick some quiet space after that  
        setSampleValueAt(target, targetIndex, 0)  
        targetIndex = targetIndex + 1  
    play(target) # Let's hear and return the result  
    return target
```

# What's going on here?

- First, set up a source and target.
- Next, we copy “United” (samples 33414 to 40052) after “We the” (sample 17408)
  - That means that we end up at  $17408 + (40052 - 33414) = 17408 + 6638 = 24046$
  - Where does “People” start?
- Next, we copy “People” (17408 to 26726) immediately afterward.
  - Do we have to copy “of” to?
  - Or is there a pause in there that we can make use of?
- Finally, we insert a little ( $1/1441^{\text{th}}$  of a second) of space – 0's

Word	Ending index
We	15730
the	17407
People	26726
of	32131
the	33413
United	40052
Kingdom	55510

# What if we didn't do that second copy? Or the pause?

```
def spliceSimpler():  
    file = getMediaPath("preamble10.wav")  
    source = makeSound(file)  
    target = makeSound(file) # This will be the newly spliced sound  
    targetIndex = 17408 # targetIndex starts at just after "We the" in the new sound  
    for sourceIndex in range(33414, 40052): # Where the word "United" is in the sound  
        setSampleValueAt(target, targetIndex, getSampleValueAt(source, sourceIndex))  
        targetIndex = targetIndex + 1  
  
    # Let's hear and return the result  
    play(target)  
    return target
```

# General clip function

We can simplify those splicing functions if we had a general clip method that took a start and end index and returned a new sound clip with just that part of the original sound in it.

```
def clip(source, start, end):  
    target = makeEmptySound(end - start)  
    tIndex = 0  
    for sIndex in range(start, end):  
        value = getSampleValueAt(source, sIndex)  
        setSampleValueAt(target, tIndex, value)  
        tIndex = tIndex + 1  
    return target
```

# General copy function

We can also simplify splicing if we had a general copy method that took a source and target sounds and copied the source into the target starting at a specified target location.

```
def copy(source, target, start):  
    tIndex = start  
    for sIndex in range(0, getLength(source)):  
        value = getSampleValueAt(source, sIndex)  
        setSampleValueAt(target, tIndex, value)  
        tIndex = tIndex + 1
```

# Simplified preamble splice

Now we can use these functions to insert “United” into the preamble in a much simpler way.

```
def createNewPreamble():
    file = getMediaPath("preamble10.wav")
    preamble = makeSound(file)      # old preamble
    united = clip(preamble, 33414, 40052) # "United"
    start = clip(preamble, 0, 17407)   # "We the"
    end = clip(preamble, 17408, 55510) # the rest
    len = getLength(start) + getLength(united)
    len = len + getLength(end) # length of everything
    newPre = makeEmptySound(len)      # new preamble
    copy(start, newPre, 0)
    copy(united, newPre, getLength(start))
    copy(end, newPre, getLength(start)+getLength(united))
    return newPre
```

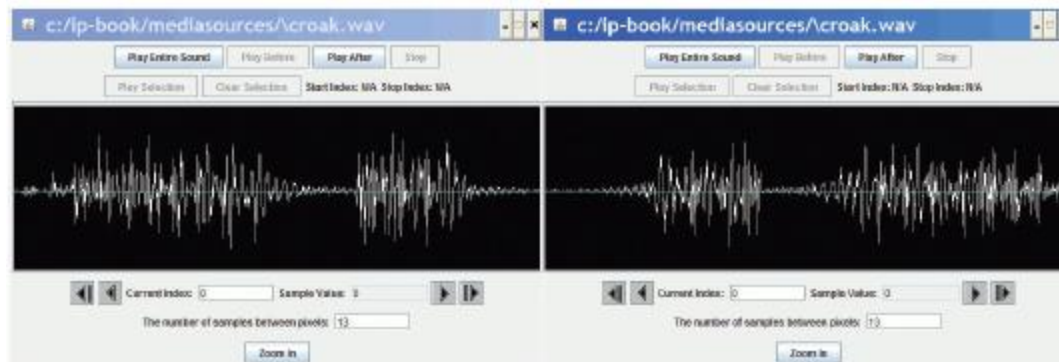
# Changing the splice

- What if we wanted to increase or decrease the volume of an inserted word?
  - Simple! Multiply each sample by something as it's pulled from the source.
- Could we do something like slowly increase volume (emphasis) or normalize the sound?
  - Sure! Just like we've done in past programs, but instead of working across *a*//samples, we work across only the samples in that sound!

# Reversing Sounds

- We can also modify sounds by reversing them

```
def reverse(source):  
    target = makeEmptySound(getLength(source))  
    sourceIndex = getLength(source) - 1 # start at end  
    for targetIndex in range(0, getLength(target)):  
        value = getSampleValueAt(source, sourceIndex)  
        setSampleValueAt(target, targetIndex, value)  
        sourceIndex = sourceIndex - 1 # move backwards  
    return target
```





# What does `makeEmptySong` take as input?

Based on that last program, what do you think `makeEmptySong` takes as input?

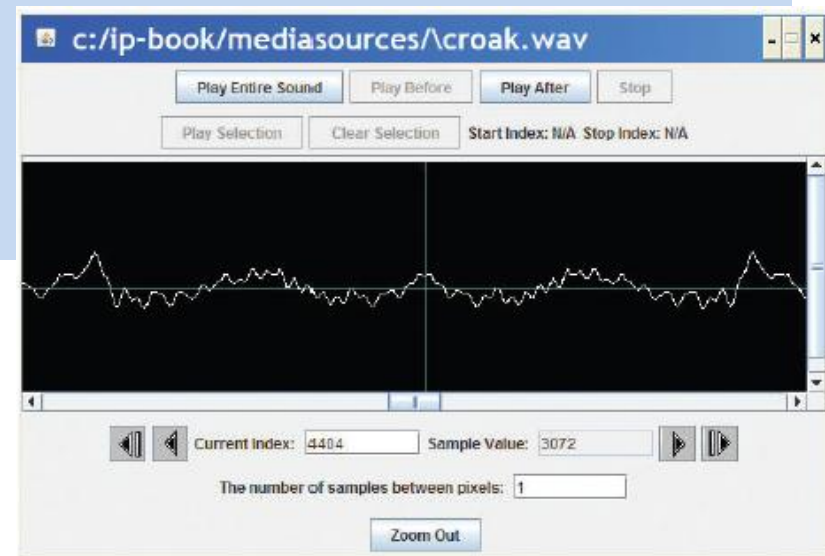
1. Number of samples needed in the new song.
2. Number of bytes needed in the new sound.
3. Number of seconds needed in the new song.
4. A song to copy.



# Mirroring

- We can mirror sounds in exactly the same way we mirrored pictures

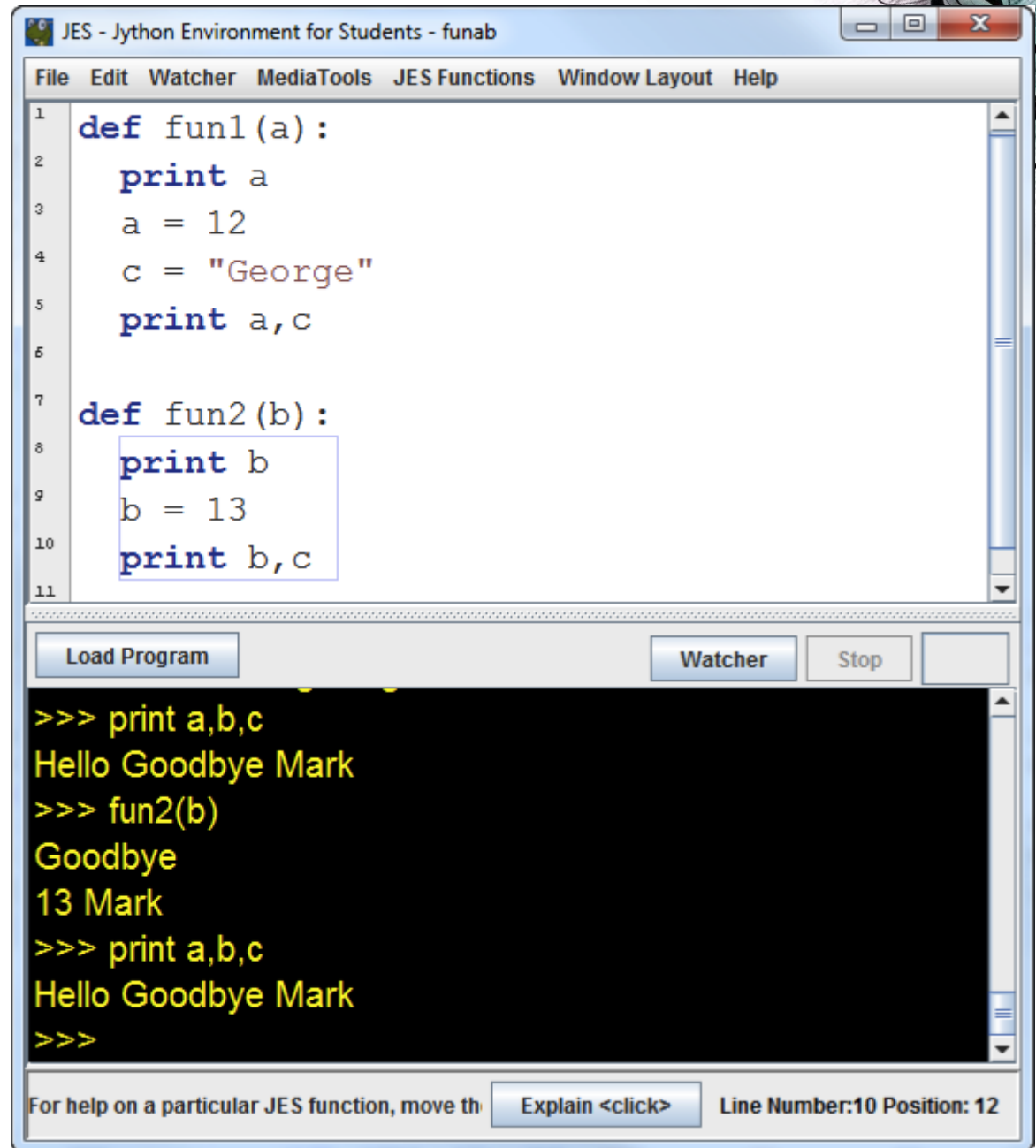
```
def mirrorSound(sound):  
    len = getLength(sound)  
    mirrorpoint = len/2  
    for index in range(0, mirrorpoint):  
        left = getSampleObjectAt(sound, index)  
        right = getSampleObjectAt(sound, len-index-1)  
        value = getSampleValue(left)  
        setSampleValue(right, value)
```



# Functions and Scope

- Defined:
  - Let's call the variable that represents the input a "parameter variable"
- Key idea:
  - The parameter variable in a function has *NOTHING* to do with any variable (even with the same name) in the Command Area – or anywhere else.
- Parameter variables are *LOCAL* to the function.
  - We say that it's in the function's *SCOPE*.

Think this  
through:



The screenshot shows the JES IDE window titled "JES - Jython Environment for Students - funab". The menu bar includes File, Edit, Watcher, MediaTools, JES Functions, Window Layout, and Help. The editor displays a Python script with two functions: `fun1(a)` and `fun2(b)`. `fun1` prints `a`, sets `a = 12`, sets `c = "George"`, and prints `a, c`. `fun2` prints `b`, sets `b = 13`, and prints `b, c`. The output window shows the execution results: `>>> print a,b,c` results in "Hello Goodbye Mark", `>>> fun2(b)` results in "Goodbye 13 Mark", and `>>> print a,b,c` results in "Hello Goodbye Mark". The status bar at the bottom indicates "Line Number:10 Position: 12".

```
1 def fun1(a):
2     print a
3     a = 12
4     c = "George"
5     print a,c
6
7 def fun2(b):
8     print b
9     b = 13
10    print b,c
11
```

Load Program Watcher Stop

```
>>> print a,b,c
Hello Goodbye Mark
>>> fun2(b)
Goodbye
13 Mark
>>> print a,b,c
Hello Goodbye Mark
>>>
```

For help on a particular JES function, move th Explain <click> Line Number:10 Position: 12

# Values are copied into parameters

- When a function is called, the input values are copied into the parameter variables.
  - Changing the parameter variables can't change the input variables.
- All variables that are local disappear at the end of the function.
- We can reference variables external to the function, if we don't have a local variable with the same name.

# Parameters as Objects

- *Note:* Slightly different when you pass an object, like a Sound or a Picture.
  - You still can't change the original *variable*, but you've passed in the object. You can change the object.

```
>>> p = makePicture(pickAFile())
```

```
>>> increaseRed(p)
```

- `increaseRed()` can't change the variable `p`, but it can apply functions and methods to change the *picture* that `p` references.
- That picture, the object, is the *value* that we passed in to the function.



Introduction to Digital Sound

# FINAL REMARKS

# Additional Resources

- **Royalty-Free Digital Sound Clips**

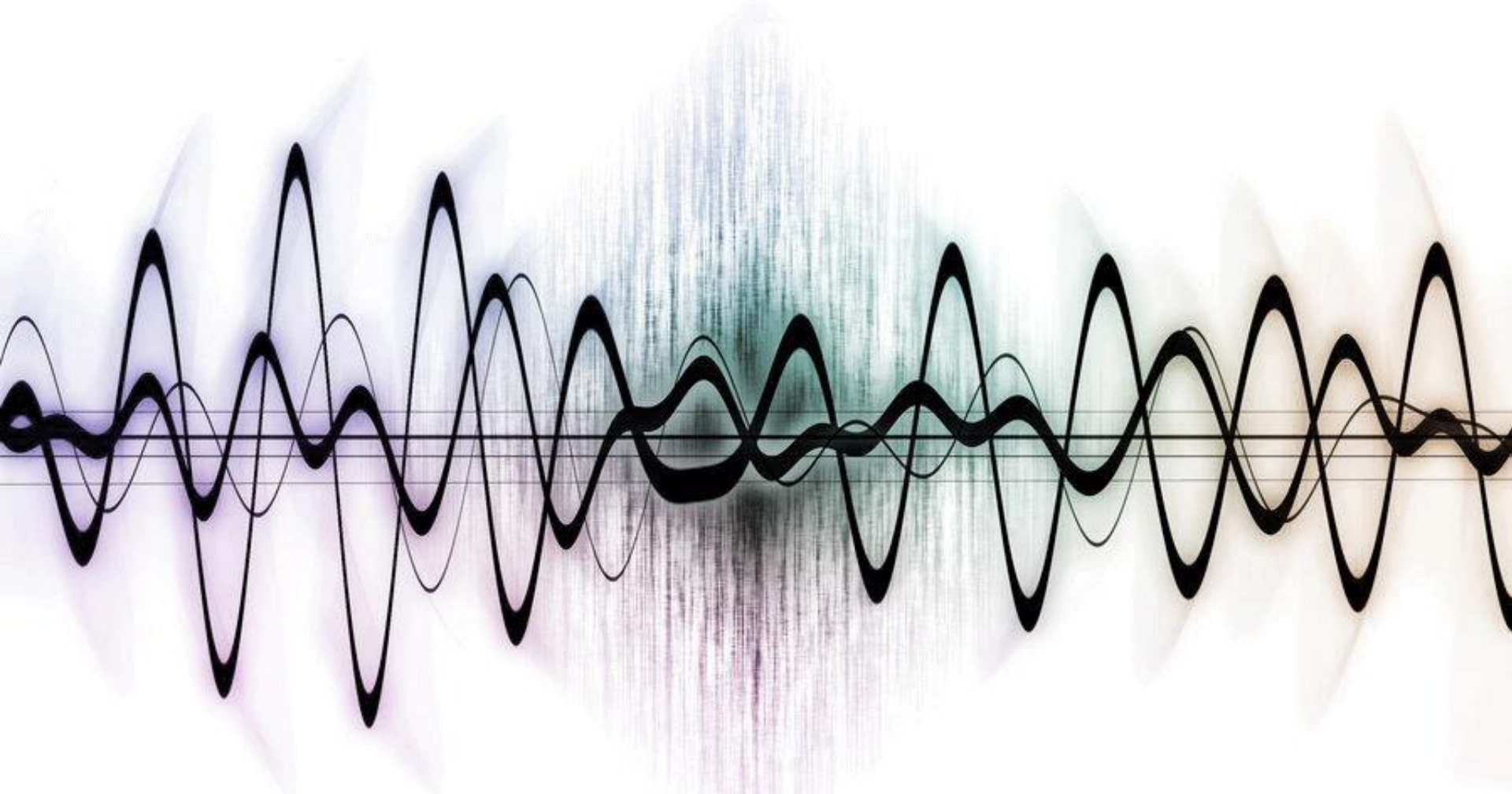
<http://incompetech.com/>

- **Digital Sound Manipulation**

<http://www.superflashbros.net/as3sfxr/>







**Thank You For Listening**

Michael Scott