Introduction to Computing and Programming in Python:

A Multimedia Approach

Chapter 6: Modifying Sounds Using Loops

Chapter Objectives

The media learning goals for this chapter are:

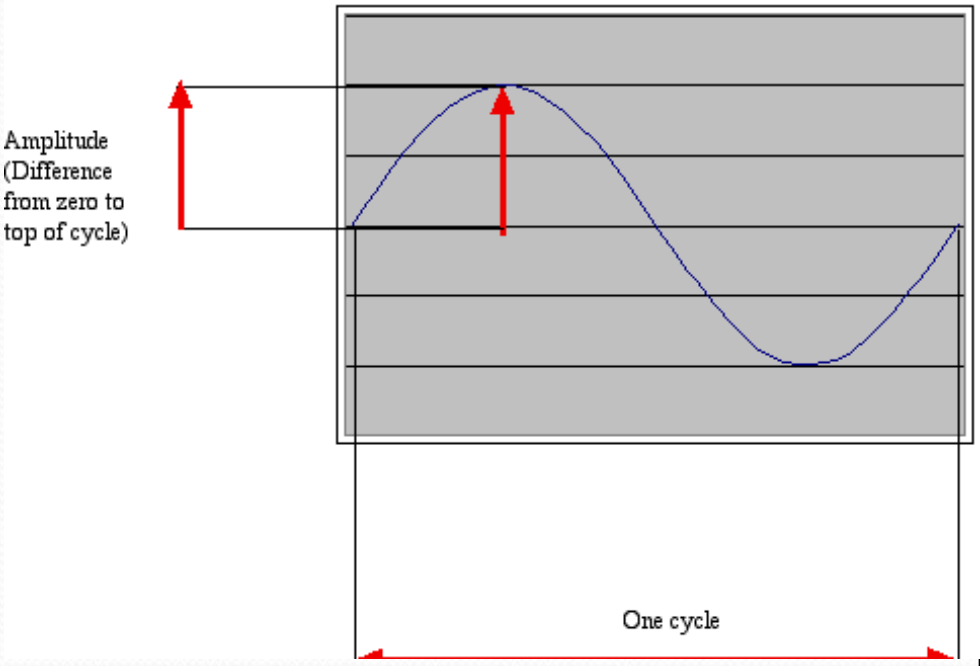
- To understand how we digitize sounds, and the limitations of human hearing that allow us to digitize sounds.
- To use the Nyquist theorem to determine the sampling rate necessary for digitizing a desired sound.
- To manipulate volume.
- To create (and avoid) clipping.

The computer science goals for this chapter are:

- To understand and use arrays as a data structure.
- To use the formula that *n* bits result in 2^n possible patterns in order to figure out the number of bits needed to save values.
- To use the sound object.
- To debug sound programs.
- To use iteration (in `for` loops) for manipulating sounds.
- To use scope to understand when a variable is available for us.

How sound works: Acoustics, the physics of sound

- 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.
  - The amplitude is the maximum height of the wave



Volume and Pitch: Psychoacoustics, the psychology of sound

- 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

“Logarithmically?”

- It’s strange, but our hearing works on ratios not differences, e.g., for pitch.
  - We hear the difference between 200 Hz and 400 Hz, as the same as 500 Hz and 1000 Hz
  - Similarly, 200 Hz to 600 Hz, and 1000 Hz to 3000 Hz
- Intensity (volume) is measured as watts per meter squared
  - A change from 0.1W/m2 to 0.01 W/m2, sounds the same to us as 0.001W/m2 to 0.0001W/m2

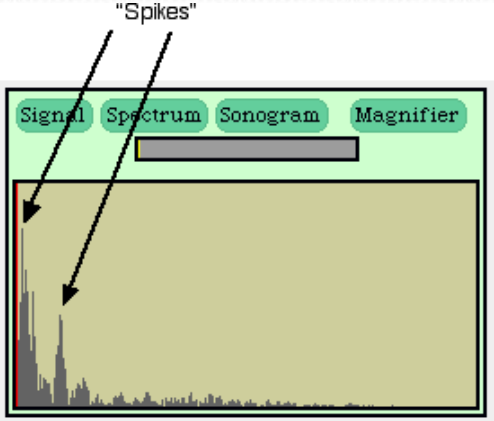
Decibel is a logarithmic measure

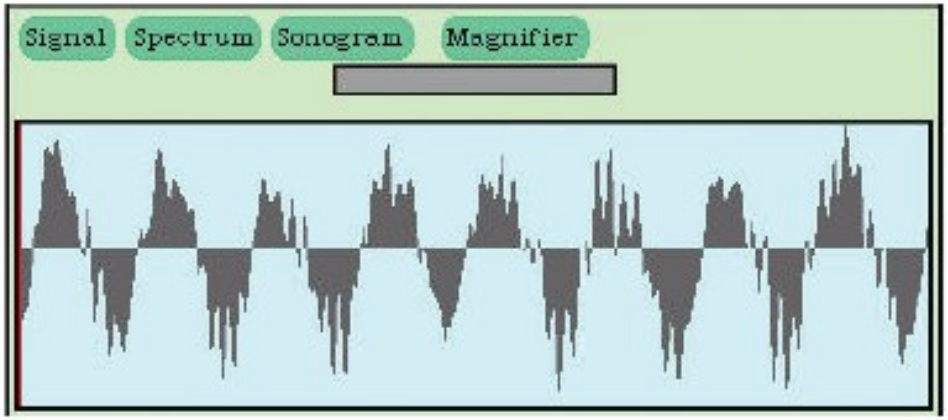
- A decibel is a ratio between two intensities:
  `10 * log10(I1/I2)`
  - As an absolute measure, it's in comparison to threshold of audibility
  - 0 dB can't be heard.
  - Normal speech is 60 dB.
  - A shout is about 80 dB

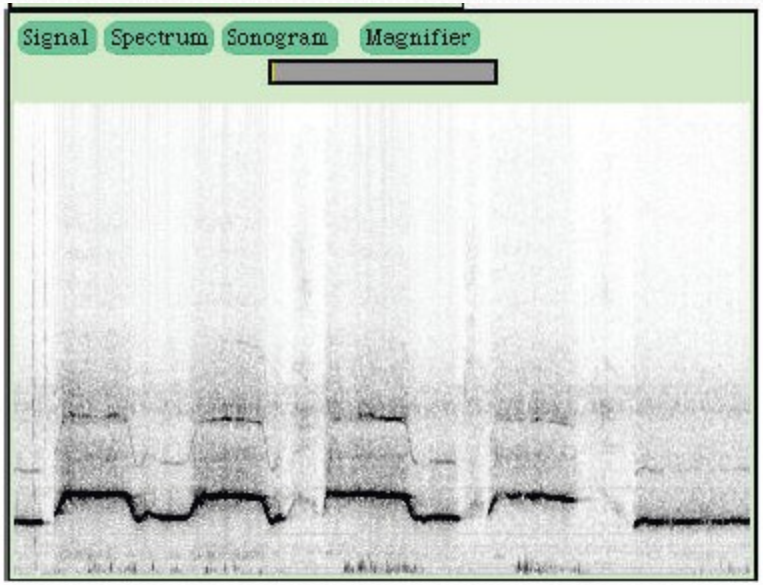
Demonstrating Sound MediaTools

Click here to see viewers while recording

Fourier transform (FFT)

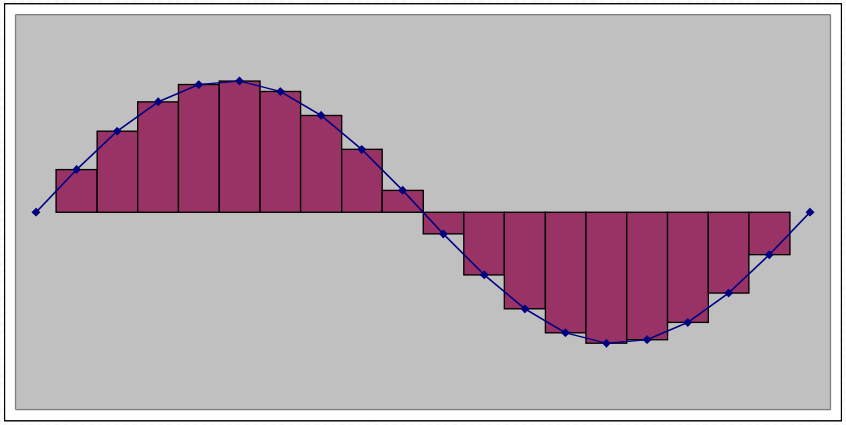






Digitizing Sound: How do we get that into numbers?

- Remember in calculus, estimating the curve by creating rectangles?
- We can do the same to estimate the sound curve
  - Analog-to-digital conversion (ADC) will give us the amplitude at an instant as a number: a sample
  - How many samples do we need?



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

Digitizing sound in the computer

- Each sample is stored as a number (two bytes)
- What's the range of available combinations?
  - 16 bits, 216 = 65,536
  - But we want both positive and negative values
    - To indicate compressions and rarefactions.
  - What if we use one bit to indicate positive (0) or negative (1)?
  - That leaves us with 15 bits
  - 15 bits, 215 = 32,768
  - One of those combinations will stand for zero
    - We'll use a "positive" one, so that's one less pattern for positives

Two's Complement Numbers

011	+3	Imagine there are only 3 bits
010	+2	we get $2^3 = 8$ possible values
001	+1	Subtracting 1 from 2 we borrow 1
000	0	
111	-1	Subtracting 1 from 0 we borrow 1's
110	-2	which turns on the high bit for all
101	-3	negative numbers
100	-4	

Two's complement numbers can be simply added

Adding -9 (11110111) and 9 (00001001)

$$
\begin{array}{r}
1111111 \\
\hline
00001001 \\
+\,11110111 \\
\hline
00000000
\end{array}
$$

+/- 32K

- Each sample can be between -32,768 and 32,767

Why such a bizarre number?

Because $32,768 + 32,767 + 1 = 2^{16}$

< 0

> 0

0

i.e. 16 bits, or 2 bytes

Compare this to 0...255 for light intensity

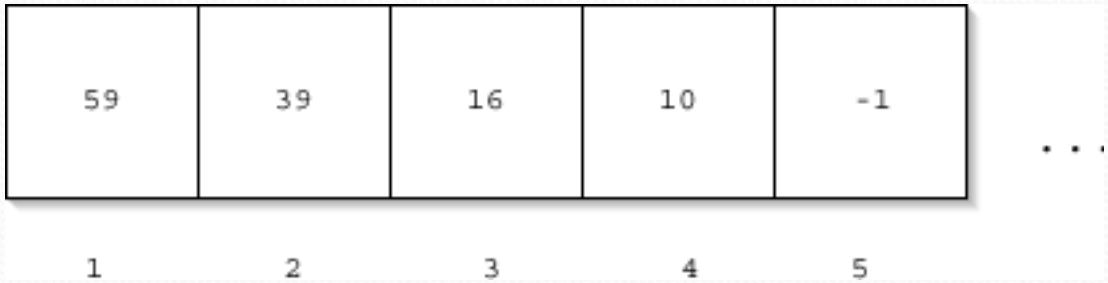
(i.e. 8 bits or 1 byte)

Sounds as arrays

- Samples are just stored one right after the other in the computer's memory

(Like pixels in a picture)

- That's called an array
  - It's an especially efficient (quickly accessed) memory structure

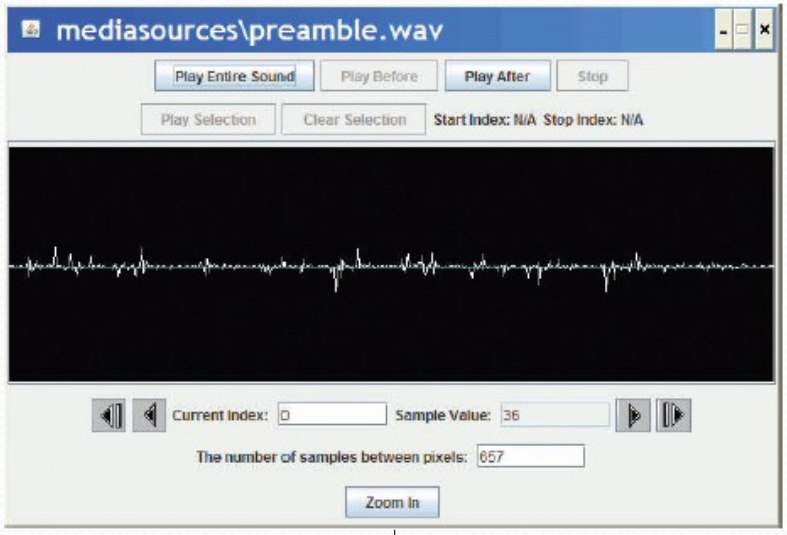


Working with sounds

- We'll use **pickAFile** and **makeSound**.
  - We want .wav files
- We'll use **getSamples** to get all the *sample objects* out of a sound
- We can also get the value at any index with **getSampleValueAt**
- Sounds also know their length (**getLength**) and their sampling rate (**getSamplingRate**)
- Can save sounds with **writeSoundTo(sound, "file.wav")**

Demonstrating Working with Sound in JES

```
>>> filename=pickAFile()
>>> print filename
/Users/guzdial/mediasources/preamble.wav
>>> sound=makeSound(filename)
>>> print sound
Sound of length 421109
>>> samples=getSamples(sound)
>>> print samples
Samples, length 421109
>>> print getSampleValueAt(sound,1)
36
>>> print getSampleValueAt(sound,2)
29
>>> explore(sound)
```



Demonstrating working with samples

```
>>> print getLength(sound)
220568
>>> print getSamplingRate(sound)
22050.0
>>> print getSampleValueAt(sound,220568)
68
>>> print getSampleValueAt(sound,220570)
I wasn't able to do what you wanted.
The error java.lang.ArrayIndexOutOfBoundsException has occurred
Please check line 0 of
>>> print getSampleValueAt(sound,1)
36
>>> setSampleValueAt(sound,1,12)
>>> print getSampleValueAt(sound,1)
12
```

Working with Samples

- We can get sample objects out of a sound with **getSamples**(**sound**) or **getSampleObjectAt**(**sound,index**)
- A sample object remembers its sound, so if you change the sample object, the sound gets changed.
- Sample objects understand **getSample**(**sample**) and **setSample**(**sample,value**)

Example: Changing Samples

```
>>> soundfile=pickAFile()
>>> sound=makeSound(soundfile)
>>> sample=getSampleObjectAt(sound,1)
>>> print sample
Sample at 1 value at 59
>>> print sound
Sound of length 387573
>>> print getSound(sample)
Sound of length 387573
>>> print getSample(sample)
59
>>> setSample(sample,29)
>>> print getSample(sample)
29
```

“But there are thousands of these samples!”

- How do we do something to these samples to manipulate them, when there are thousands of them per second?
- We use a loop and get the computer to iterate in order to do something to each sample.
- An example loop:

```
for sample in getSamples(sound):
    value = getSample(sample)
    setSample(sample,value)
```

Recipe to Increase the Volume

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

How did that work?

- When we evaluate increaseVolume(s), the function increaseVolume is executed
- The sound in variable s becomes known as sound
- Sound is a placeholder for the sound object s.

```
>>> f=pickAFile()
>>> s=makeSound(f)
>>> increaseVolume(s)
```

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

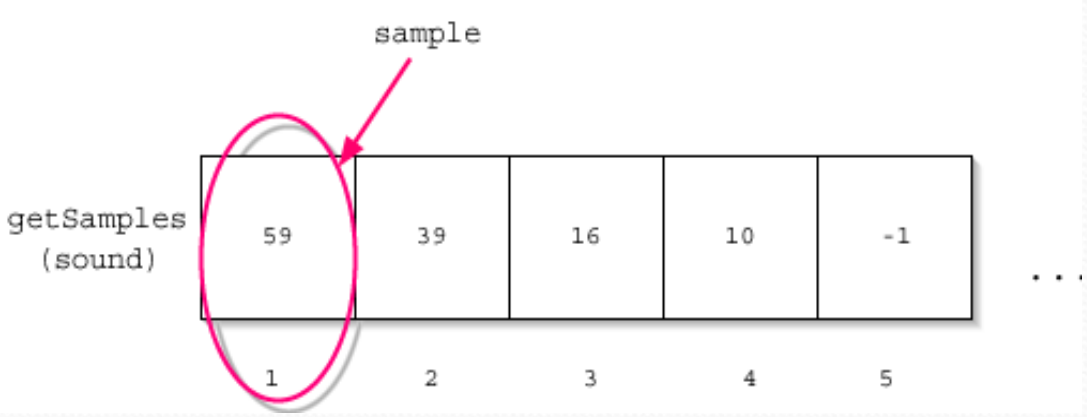
Starting the loop

- **getSamples(sound)** returns a sequence of all the sample objects in the **sound**.
- The **for** loop makes **sample** be the first sample as the block is started.

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

Compare:

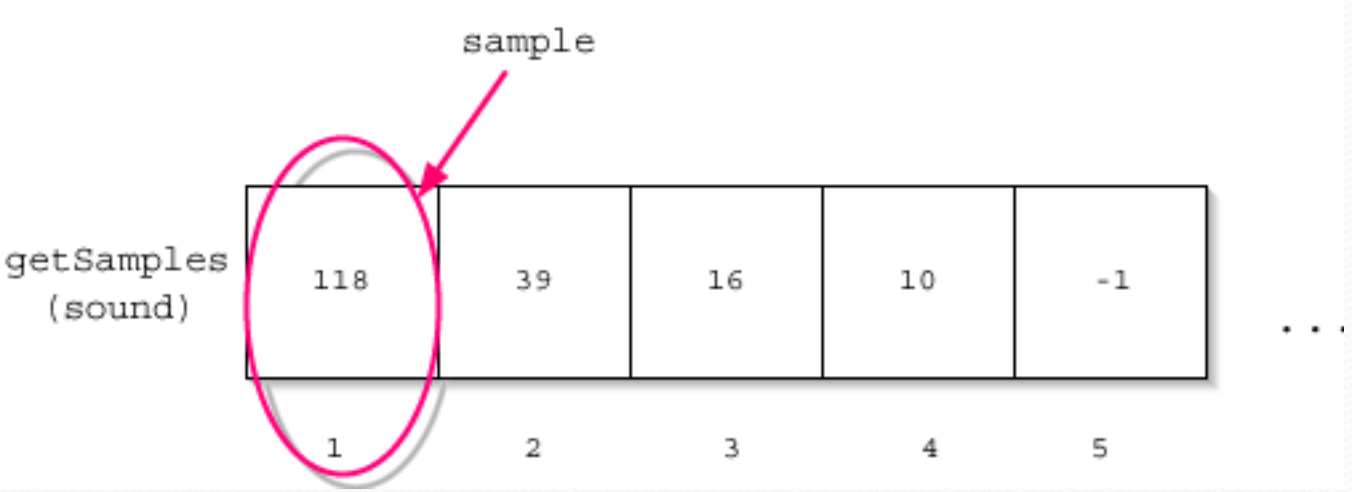
```
for pixel in
getPixels(picture):
```



Executing the block

- We get the value of the sample named **sample.**
- We set the value of the sample to be the current value (variable **value**) times 2

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



Next sample

- Back to the top of the loop, and **sample** will now be the second sample in the sequence.

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

And increase that next sample

- We set the value of *this* sample to be the current value (variable **value**) times 2.

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

And on through the sequence

- The loop keeps repeating until *all* the samples are doubled

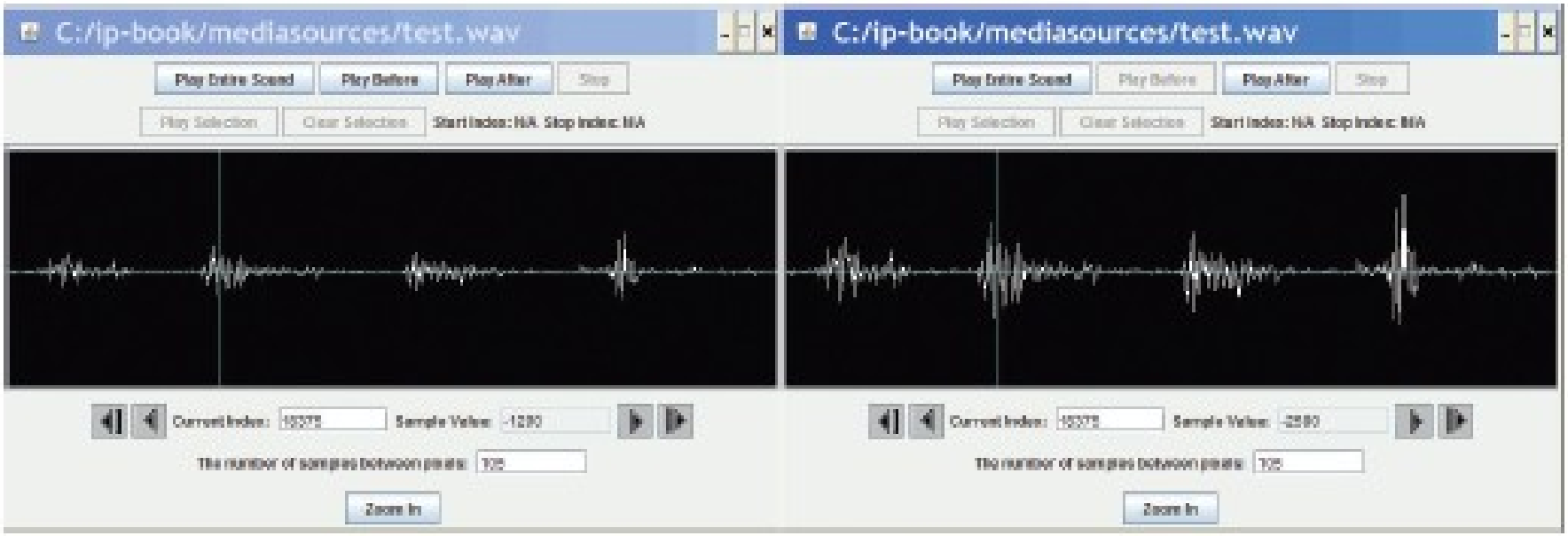
```
def increaseVolume(sound):
  for sample in
getSamples(sound):
    value =
getSampleValue(sample)
    setSampleValue(sample,value
* 2)
```

How are we *sure* that that worked?

```
>>> print s
Sound of length 220567
>>> print f
/Users/guzdial/mediasources/gettysburg10.wav
>>> soriginal=makeSound(f)
>>> print getSampleValueAt(s,1)
118
>>> print getSampleValueAt(soriginal,1)
59
>>> print getSampleValueAt(s,2)
78
>>> print getSampleValueAt(soriginal,2)
39
>>> print getSampleValueAt(s,1000)
-80
>>> print getSampleValueAt(soriginal,1000)
-40
```

Here we're comparing the modified sound `s` to a copy of the original sound `soriginal`

Exploring both sounds



The right side does *look* like it's larger.

Decreasing the volume

```
def decreaseVolume(sound):
    for sample in
getSamples(sound):
        value =
getSampleValue(sample)
        setSampleValue(sample,value
* 0.5)
```

This works *just* like **increaseVolume**, but we're *lowering* each sample by 50% instead of doubling it.

We can make this generic

- By adding a *parameter*, we can create a general **changeVolume** that can increase *or* decrease volume.

```
def changeVolume(sound ,
factor):
  for sample in
getSamples(sound):
    value =
getSampleValue(sample)
    setSampleValue(sample
,value * factor)
```

Recognize some similarities?

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

```
def increaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*1.2)
```

```
def decreaseVolume(sound):
  for sample in
getSamples(sound):
    value =
getSampleValue(sample)
    setSampleValue(sample,
value*0.5)
```

```
def decreaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*0.5)
```

Does increasing the volume change the volume setting?

- No
  - The physical volume setting indicates an upper bound, the potential loudest sound.
  - Within that potential, sounds can be louder or softer
    - They can fill that space, but might not.

(Have you ever noticed how commercials are always louder than regular programs?)

- **Louder content attracts your attention.**
- **It maximizes the *potential* sound.**

Maximizing volume

- How, then, do we get maximal volume?
  - (e.g. automatic recording level)
- It's a three-step process:
  - First, figure out the loudest sound (largest sample).
  - Next, figure out how much we have to increase/decrease that sound to fill the available space
    - We want to find the amplification factor amp, where amp * loudest = 32767
    - In other words: amp = 32767/loudest
  - Finally, amplify each sample by multiplying it by amp

Maxing (*normalizing*) the sound

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

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

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

Max()

- **max**() is a function that takes *any* number of inputs, and always returns the largest.
- There is also a function **min**() which works similarly but returns the minimum

```
>>> print max(1,2,3)
3
>>> print max(4,67,98,-1,2)
98
```

Or: use if instead of max

```
def normalize(sound):
  largest = 0
  for s in getSamples(sound):
    if getSampleValue(s) > largest:
      largest = getSampleValue(s)
  amplification = 32767.0 / largest
  print "Largest sample value in original sound was",
 largest
  print ”Amplification factor is", amplification
  for s in getSamples(sound):
    louder =  amplification * getSampleValue(s)
    setSampleValue(s, louder)
```

Aside: positive and negative extremes assumed to be equal

- We're making an assumption here that the maximum positive value is also the maximum negative value.
  - That should be true for the sounds we deal with, but isn't necessarily true
- Try adding a constant to every sample.
  - That makes it non-cyclic
    - I.e. the compressions and rarefactions in the sound wave are not equal
  - But it's fairly subtle what's happening to the sound.

Why 32767.0, not 32767?

- Why do we divide out of 32767.0 and not just simply 32767?
  - Because of the way Python handles numbers
  - If you give it integers, it will only ever compute integers.

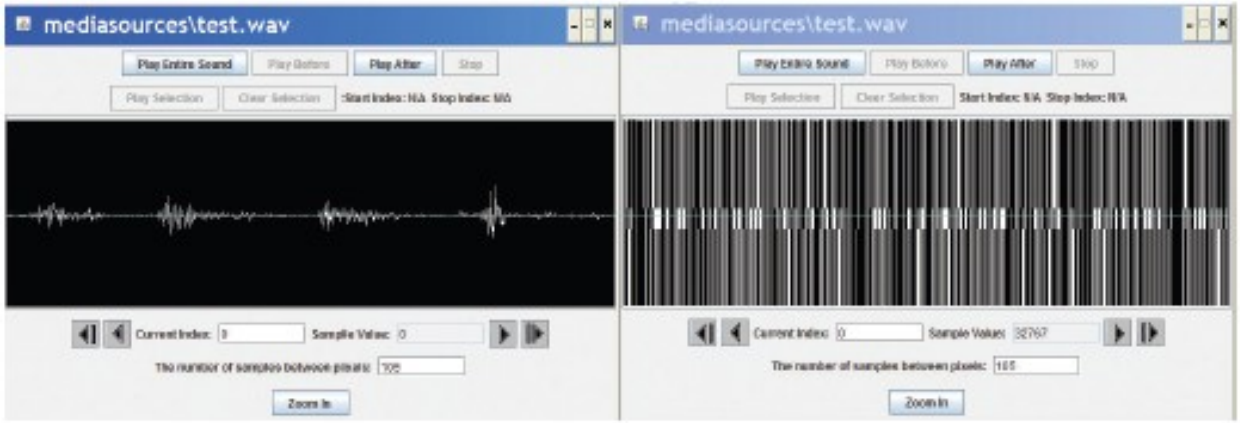
```
>>> print 1.0/2
0.5
>>> print 1.0/2.0
0.5
>>> print 1/2
0
```

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



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()