## Sound Manipulations

In the previous section, we discussed how to change the volume of a sound by modifying the values of the amplitude stored in each sample. To increase the volume, we multiply each sample value by a factor greater than 1, and to decrease the volume, we multiply each sample value by a factor less than 1. This is very similar to how we increased and decreased the red, green, and blue color values in a picture. With pictures, we saw that we did not need separate functions to increase and decrease the red/green/blue values: we could use one function that takes the multiplier as a parameter. The same idea applies to volume – we don't need separate functions to increase and decrease it. We can use one function that takes the multiplier as a parameter.

**Example:** ChangeVolume

```
# This function changes the volume of a sound
# A factor > 1 increases the volume
# A factor < 1 decreases the volume
def changeVolume(sound, factor):
  newSound = duplicateSound(sound)
  for sample in getSamples(newSound):
    value = getSampleValue(sample)
    setSampleValue(sample, value * factor)
  return newSound
```

We also previously discussed that if the volume of a sound is increased too much, we will get the effect of clipping taking place. But now, suppose we want to make the volume of a sound to be as loud as possible. This is called *normalizing* the sound. In order to do this, we need to find out what the largest (absolute) value of a sample in the sound is. Once we know the largest value, we figure out what factor to multiply our samples by. We know the largest value a sample can have is 32767. If we divide this by our largest sample value, we get the factor we should use:

$$factor = \frac{32767}{largest\ sample\ value}.$$

To find the largest absolute value of the samples, we will define a variable, say, `largest`, and assign it to be 0. (Using absolute values, every value will be greater than or equal to 0). Then we will check all the sample values. If we find a sample value with absolute value greater than `largest`, we will replace `largest` with that new value. We will keep checking the sample values, comparing to the new value of `largest`, until we have compared all sample values, and the very largest value is stored in the variable `largest`. Python has a built-in function called `max` that will figure out the maximum of two values. We will use this to help us. Our function to normalize a sound looks like the following:

**Example:** Normalize a sound
```
# This function makes a sound as loud as possible
# without clipping
def normalize(sound):
  newSound = duplicateSound(sound)

  # find the largest sample value
  largest = 0
  for s in getSamples(newSound):
    largest = max(largest, abs(getSampleValue(s)))

  # compute the multiplication factor
  factor = 32767.0/largest

  # change the sample values
  for s in getSamples(newSound):
    value = getSampleValue(s)
    setSampleValue(s, value * factor)

  return newSound
```

**Exercise:** Notice that the last loop in the `normalize` function looks exactly like the `changeVolume` function. Rewrite the `normalize` function so that it calls the `changeVolume` function.

**Exercise:** Write a function called `onlyMaximize` that takes a sound as input and will set all sample values to be the maximum values. This function should duplicate the original sound and make all changes to the copy. If a sample value is positive, it should be set to 32767, and if it is negative, it should be set to -32768. The new sound should be returned.

**Follow-up questions:** What does `onlyMaximize` do to a sound? What do you hear?

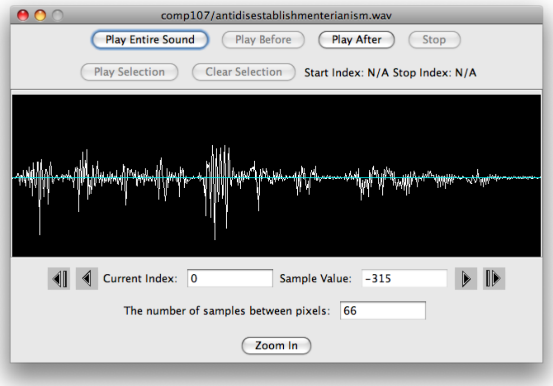
In the previous section, we saw that there are two ways to loop through all of the samples in a sound – we can use the `getSamples` function , or we can use the `range` function with `getNumSamples`. When we use the latter method, we work with the index of each sample to get and set the values. This method will give us more freedom to access only parts of the sound, instead of the entire sound.

Why might this be useful? Maybe we only want to change the volume of part of a sound, or maybe we want to change the frequency of part of a sound, or maybe we want to reverse a sound or part of a sound.

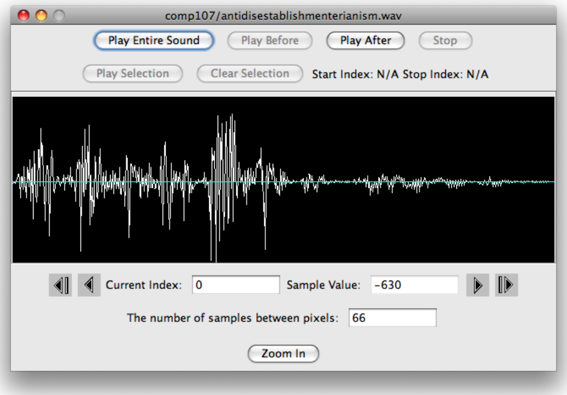
In the following example, we double the volume of the first half of the sound and decrease the volume by 40% in the second half.

**Example:** Increase and Decrease

```
def increaseAndDecrease(sound):
  newSound = duplicateSound(sound)
  numSamples = getNumSamples(newSound)

  # double volume in first half
  for index in range(numSamples/2):
    value = getSampleValueAt(newSound, index)
    setSampleValueAt(newSound, index, value * 2)

  # decrease the volume of second half by 40%
  for index in range(numSamples/2, numSamples):
    value = getSampleValueAt(newSound, index)
    setSampleValueAt(newSound, index, value * 0.60)

  # return the new sound
  return newSound
```


comp107/antidisestablishmenterianism.wav
Play Entire Sound
Play Before
Play After
Stop
Play Selection
Clear Selection
Start Index: N/A Stop Index: N/A
Current Index: 0
Sample Value: -315
The number of samples between pixels: 66
Zoom In

Original Sound

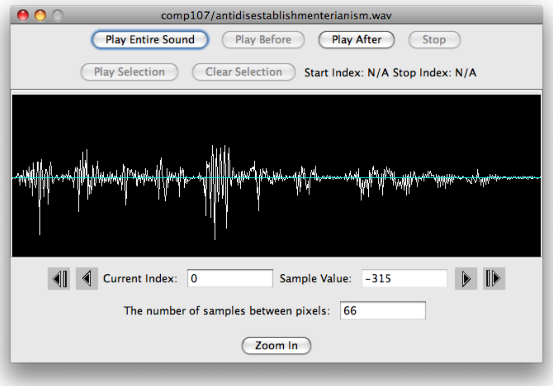

comp107/antidisestablishmenterianism.wav
Play Entire Sound
Play Before
Play After
Stop
Play Selection
Clear Selection
Start Index: N/A Stop Index: N/A
Current Index: 0
Sample Value: -630
The number of samples between pixels: 66
Zoom In

Result after increase and decrease

Another example where using the indices of sounds is useful is in reversing a sound. To reverse a sound, we want to copy the samples in one sound in reverse order to a new sound. Here's an example of how we might do this:

**Example:** Reversing a sound

```
def reverse(sound):
  # create a new empty sound with same # of samples as
  # the original sound
  newSound = makeEmptySound(getNumSamples(sound))
  # set up index to start at end of new sound
  newIndex = getNumSamples(sound)-1
  # loop through original sound, setting
  # values in new sound
  for index in range(getNumSamples(sound)):
    value = getSampleValueAt(sound, index)
    setSampleValueAt(newSound, newIndex, value)
    newIndex = newIndex -1

  # return the new sound
  return newSound
```



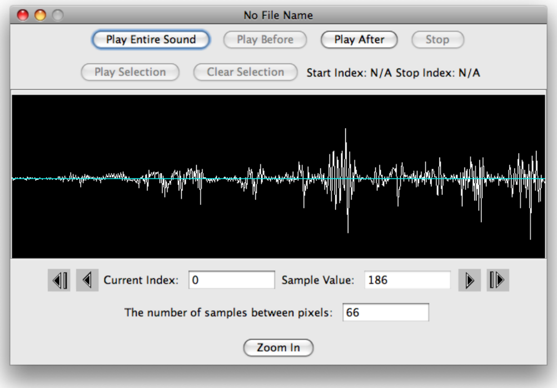
comp107/antidisestablishmenterianism.wav

Start Index: N/A Stop Index: N/A

Current Index: 0 Sample Value: -315

The number of samples between pixels: 66

original sound



No File Name

Start Index: N/A Stop Index: N/A

Current Index: 0 Sample Value: 186

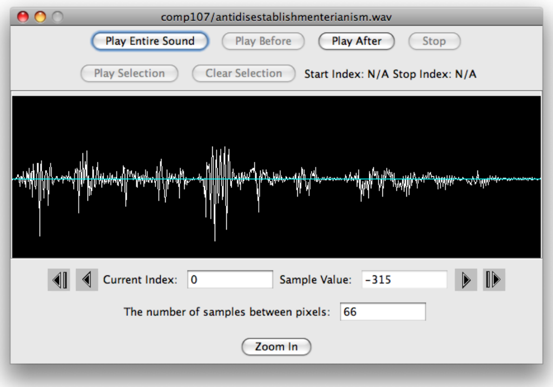
The number of samples between pixels: 66

reversed sound

**Exercise:** Write a function that mirrors the first half of the sound into the second half. That is, the second half of the sound should be the reverse of the first half.

In our introductory discussion about sound, we learned that pitch is related to frequency. We can change the pitch of a sound by changing the frequency.

In the following pictures of sounds, we can see that the original sound is being "stretched", or more precisely, the frequency has been halved. Each sample in the original sound is being used twice in the new sound. So, the length of time for one cycle has been doubled. When you look at the waves for the sounds in these pictures, you can see that the sounds have the same shape, but the second one looks stretched.

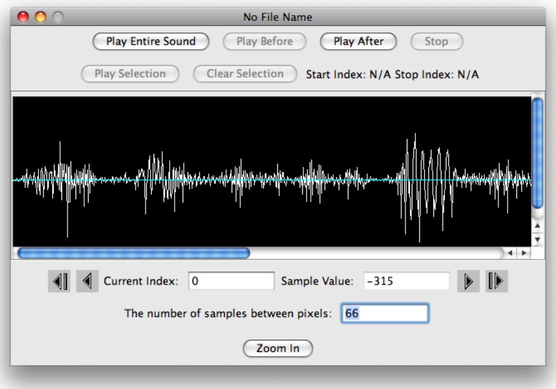


Start Index: N/A Stop Index: N/A

Current Index: 0 Sample Value: -315

The number of samples between pixels: 66

original sound



Start Index: N/A Stop Index: N/A

Current Index: 0 Sample Value: -315

The number of samples between pixels: 66

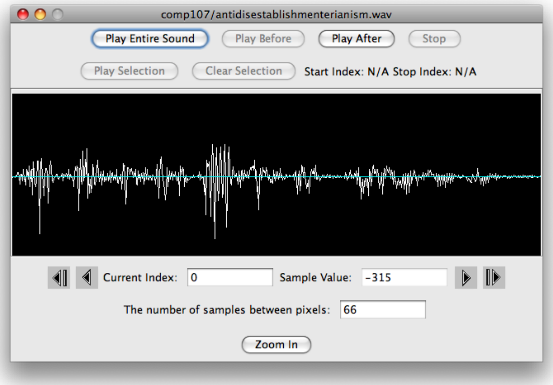
sound with halved frequency

So how do we do this with code? Our algorithm is very similar to what we did to scale a picture up in size. To make a picture larger, we used every pixel twice. We are going to do the same thing with samples in our sounds – we will use each sample value twice.

**Example:** Halving the frequency

```
# This function halves the frequency of the original sound
# The new sound is twice as long, and sounds slower and deeper
def halfFrequency(sound):
  # make a new sound with twice as many samples as original
  numSamples = getNumSamples(sound)
  newSound = makeEmptySound(numSamples * 2)

  # loop through original sound, setting values in new sound
  index = 0
  for newIndex in range(numSamples * 2):
    val = getSampleValueAt(sound, int(index))
    setSampleValueAt(newSound, newIndex, val)
    index = index + .5

  # return the new sound
  return newSound
```

We could just as easily double the frequency of a sound. Again, we would use ideas similar to what we used to make a picture smaller. To make a picture smaller, we used every other pixel; to double the frequency, we will use every other sample value. So our original sound would look like this with a doubled frequency:



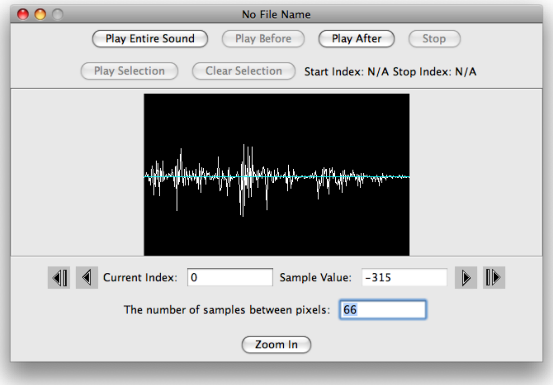
comp107/antidisestablishmentarianism.wav

Start Index: N/A Stop Index: N/A

Current Index: 0 Sample Value: -315

The number of samples between pixels: 66

original sound



No File Name

Start Index: N/A Stop Index: N/A

Current Index: 0 Sample Value: -315

The number of samples between pixels: 66

sound with doubled frequency

In the next Lab: Simple Sound Manipulation, you will experiment with these functions, as well as write a function to double the frequency of a sound. You will also experiment with splicing sounds (copying sounds into other sounds).