CHAPTER 6

Modifying Sounds Using Loops

Chapter Learning 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.

6.1 HOW SOUND IS ENCODED

There are two parts to understanding how sound is encoded and manipulated.

- First, what are the physics of sound? How is it that we hear a variety of sounds?
- Next, how can we then map sounds into numbers on a computer?

6.1.1 The Physics of Sound

Physically, sounds are waves of air pressure. When something makes a sound, it makes ripples in the air just like stones or raindrops dropped into a pond cause ripples on the surface of the water (Figure 6.1). Each drop causes a wave of pressure to pass over the surface of the water, which causes visible rises in the water, and less visible but just as large depressions in the water. The rises are increases in pressure and the lows are decreases in pressure. Some of the ripples we see actually arise from *combinations* of ripples—some waves are the sums and interactions of other waves.

In the air, we call these increases in pressure *compressions* and decreases in pressure *rarefactions*. It's these compressions and rarefactions that allow us to hear sounds. The shape of the waves, their *frequency*, and their *amplitude* all impact what we perceive in the sound.

The simplest sound in the world is a **sine wave** (Figure 6.2). In a sine wave, the compressions and rarefactions arrive with equal size and regularity. In a sine wave, one compression plus one rarefaction is called a **cycle**. At some point in the cycle, there has to be a point where there is zero pressure, just between the compression and the rarefaction. The distance from the zero point to the greatest pressure (or least pressure) is called the **amplitude**.

In general, amplitude is the most important factor in our perception of *volume*: if the amplitude rises, we typically perceive the sound as being louder. When we perceive an increase in volume, we say that we're perceiving an increase in the *intensity* of sound.

Human perception of sound is not a direct mapping from the physical reality. The study of the human perception of sound is called *psychoacoustics*. One of the odd facts about psychoacoustics is that most of our perceptions of sound are *logarithmically* related to the actual phenomena.

We measure the change in intensity in **decibels** (dB). That's probably the unit that you most often associate with volume. A decibel is a logarithmic measure, so it matches the way we perceive volume. It's always a ratio, a comparison of two values. $10 * \log_{10}(I_1/I_2)$ is the change in intensity in decibels between two intensities, I_1 and I_2. If two amplitudes are measured under the same conditions, we can express the same definition as amplitudes: $20 * \log_{10}(A_1/A_2)$. If $A_2 = 2 * A_1$ (i.e., the amplitude doubles), the difference is roughly 6 dB.

FIGURE 6.1
Raindrops causing ripples on the surface of the water, just as sound causes ripples in the air.

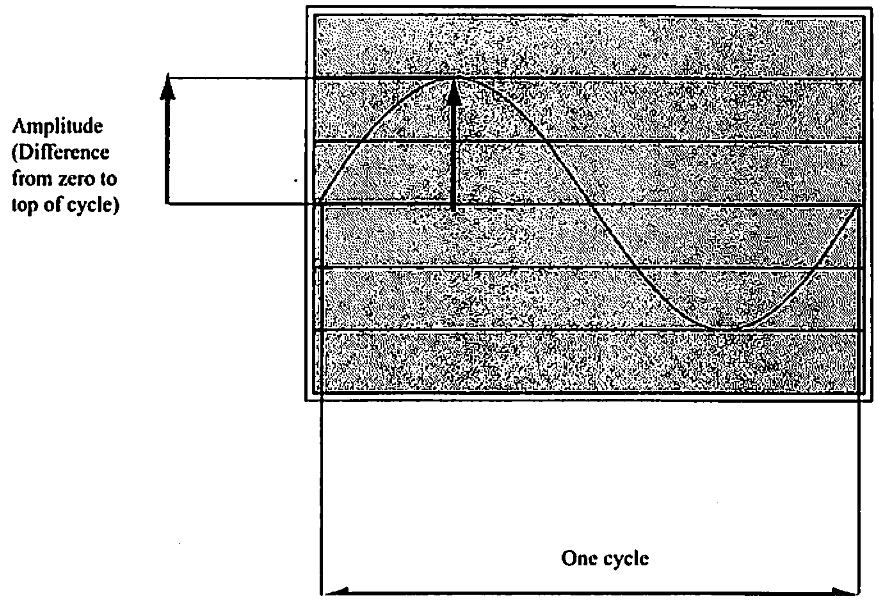

FIGURE 6.2
One cycle of the simplest sound: a sine wave.

When decibel is used as an absolute measurement, it's in reference to the threshold of audibility at *sound pressure level* (SPL): 0 dB SPL. Normal speech has an intensity of about 60 dB SPL. Shouted speech is about 80 dB SPL.

How often a cycle occurs is called the **frequency**. If a cycle is short, then there can be lots of them per second. If a cycle is long, then there are fewer of them. As the frequency increases we, perceive the **pitch** to increase. We measure frequency in *cycles per second* (cps) or *Hertz* (Hz).

All sounds are periodic—there is always some pattern of rarefaction and compression that leads to cycles. In a sine wave, the notion of a cycle is easy. In natural waves, it's not so clear where a pattern repeats. Even in the ripples in a pond, the waves aren't as regular as you might think. The time between peaks in waves isn't always the same—it varies. This means that a cycle may involve several peaks and valleys until it repeats.

Humans hear between 2 and 20,000 Hz (or 20 kilohertz, abbreviated 20 kHz). Again, as with amplitudes, that's an enormous range. To give you a sense of where music fits into that spectrum, the note A above middle C is 440 Hz in traditional *equal temperament* tuning (Figure 6.3).

Like intensity, our perception of pitch is almost exactly proportional to the log of the frequency. We don't perceive absolute differences in pitch but the *ratio* of the frequencies. If you heard a 100 Hz sound followed by a 200 Hz sound, you'd perceive the same pitch change (or *pitch interval*) as a shift from 1000 Hz to 2000 Hz. Obviously, a difference of 100 Hz is a lot smaller than a change of 1000 Hz, but we perceive it to be the same.

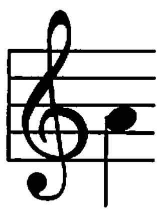

FIGURE 6.3
The note A above middle C is 440 Hz.

In standard tuning, the ratio in frequency between the same notes in adjacent octaves is 2 : 1. Frequency doubles each octave. We told you earlier that A above middle C is 440 Hz. You know, then, that the next A up the scale is 880 Hz.

How we think about music is dependent upon our cultural standards but there are some universals. Among them are the use of pitch intervals (e.g., the ratio between notes C and D remains the same in every octave), the constant relationship between octaves, and the existence of four to seven main pitches (not considering sharps and flats here) in an octave.

What makes the experience of one sound different from another? Why does a flute playing a note sound *so* different from a trumpet or a clarinet playing the same note? We still don't understand everything about psychoacoustics and what physical properties influence our perception of sound but here are some of the factors that lead us to perceive different sounds (especially musical instruments) as distinct:

- Real sounds are almost never single-frequency sound waves. Most natural sounds have *several* frequencies in them, often at different amplitudes. These additional frequencies are sometimes called *overtones*. When a piano plays the note C, for example, part of the richness of the tone is that the notes E and G are *also* in the sound, but at lower amplitudes. Different instruments have different overtones in their notes. The central tone, the one we're trying to play, is called the *fundamental*.
- Instrument sounds are not continuous with respect to amplitude and frequency. Some come slowly up to the target frequency and amplitude (like wind instruments), while others hit the frequency and amplitude very quickly and then the volume fades while the frequency remains pretty constant (like a piano).
- Not all sound waves are well represented by sine waves. Real sounds have funny bumps and sharp edges. Our ears can pick these up, at least in the first few waves. We can do a reasonable job of synthesizing with sine waves but synthesizers sometimes also use other kinds of wave forms to get different kinds of sounds (Figure 6.4).

6.1.2 Exploring How Sounds Look

At http://www.mediacomputation.org, you will find the MediaTools application with documentation for how to get it started. The MediaTools application contains tools for sound, graphics, and video. Using the sound tools, you can actually observe sounds as they come into your computer's microphone to get a sense of what louder and softer sounds look like, and what higher- and lower-pitched sounds look like.

You will also find a MediaTools menu in JES. The JES MediaTools also allow you to inspect sounds and pictures but you can't look at sounds in *real time*—as they hit your computer's microphone. You can see sounds in real time in the MediaTools application.

The MediaTools application sound editor looks like that in Figure 6.5. You can record sounds, open WAV files on your disk, and view the sounds in a variety of ways. (Of course, assuming that you have a microphone on your computer!)

To view sounds, click the Record Viewer button, then the RECORD button. (Hit the STOP button to stop recording.) There are three kinds of views that you can make of the sound.

The first is the **signal view** (Figure 6.6). In the signal view, you're looking at raw sound—each increase in air pressure results in a rise in the graph and each decrease in sound pressure results in a drop in the graph. Note how rapidly the wave changes. Try some softer and louder sounds so that you can see how their look changes. You can always get back to the signal view from another view by clicking the SIGNAL button.

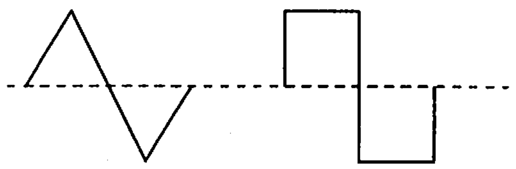

FIGURE 6.4
Some synthesizers use triangular (or *sawtooth*) or square waves.

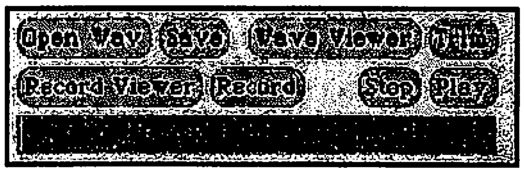

FIGURE 6.5
Sound editor main tool.

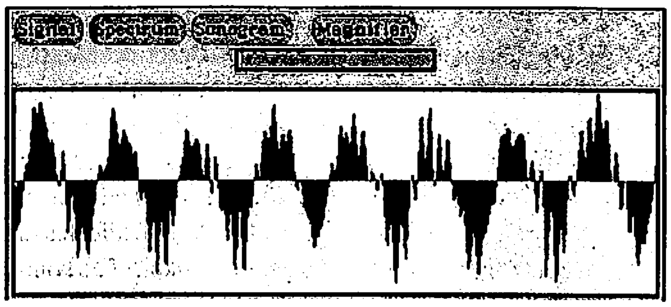

FIGURE 6.6
Viewing the sound signal as it comes in.

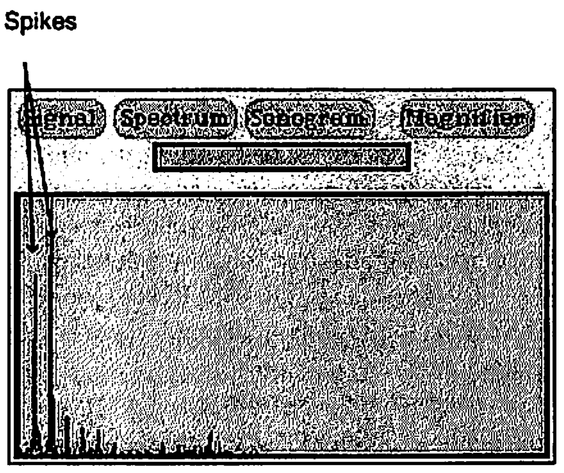

FIGURE 6.7
Viewing a sound in spectrum view with multiple spikes.

The second view is the **spectrum view** (Figure 6.7). The spectrum view is a completely different perspective on the sound. In the preceding section, you read that natural sounds are often actually composed of several different frequencies at once. The spectrum view shows these individual frequencies. This view is also called the *frequency domain.*

Frequencies increase in the spectrum view from left to right. The height of a column indicates the amount of energy (roughly, the volume) of that frequency in the sound. Natural sounds look like Figure 6.7 with more than one *spike* (rise in the graph). (The smaller rises around a spike are often seen as *noise.*)

The technical term for how a spectrum view is generated is called a **Fourier transform**. A Fourier transform takes the sound from the *time domain* (rises and falls in the sound over time) into the frequency domain (identifying which frequencies are in a sound, and the energy of those frequencies, over time). Frequencies increase in this view from left to right (leftmost are lower, rightmost are higher), and more energy at that frequency results in a taller spike.

The third view is the *sonogram view* (Figure 6.8). The sonogram view is very much like the spectrum view in that it describes the frequency domain but it presents these frequencies over time. Each column in the sonogram view, sometimes called a *slice* or *window (of time)*, represents all the frequencies at a given moment in time. The frequencies increase in the slice from lower (bottom) to higher (top). So, Figure 6.8 represents a sound that started off low and then got higher and then went lower again. The *darkness* of the spot in the column indicates the amount of energy of that frequency in the input sound at the given moment. The sonogram view is great for studying how sounds change over time (e.g., how the sound of a piano key being struck changes as the note fades, or how different instruments differ in their sounds, or how different vocal sounds differ).

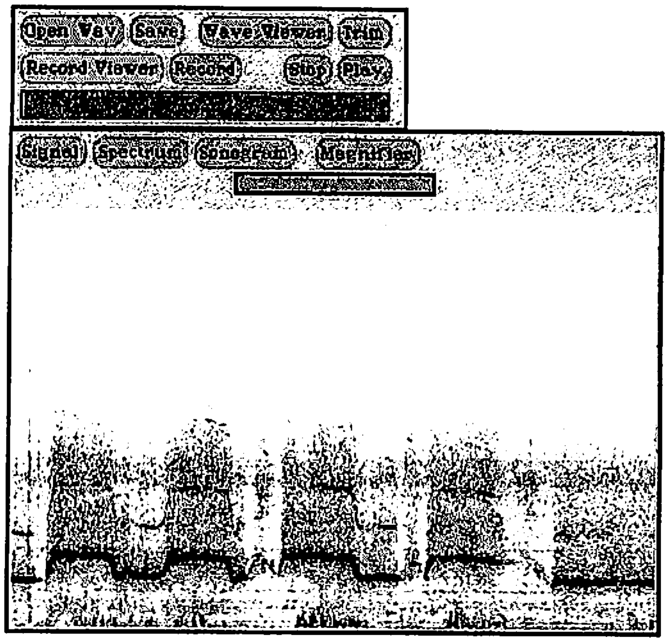

FIGURE 6.8
Viewing the sound signal in a sonogram view.

Making It Work Tip: Explore Sounds!

You really should try these different views on real sounds. You'll get a much better understanding of sound and of what the manipulations we're doing in this chapter are doing to the sounds.

■

6.1.3 Encoding the Sound

You just read about how sounds work physically and how we perceive them. To manipulate sounds on a computer and play them back on a computer, we have to digitize them. To digitize sound means to take this flow of waves and turn it into numbers. We want to be able to capture a sound, perhaps manipulate it, and then play it back (through the computer's speakers) and hear what we captured as exactly as possible.

The first part of the process of digitizing sound is handled by the computer's hardware, the physical machinery of the computer. If a computer has a microphone and appropriate sound equipment (like a SoundBlaster sound card on Windows computers), then it's possible, at any moment, to measure the amount of air pressure against the microphone as a single number. Positive numbers correspond to rises in pressure, and negative numbers correspond to rarefactions. We call this an *analog-to-digital conversion (ADC)*—we've moved from an analog signal (a continuously changing sound wave) to a digital value. This means that we can get an instantaneous measure of the sound pressure, but it's only one step along the way. Sound is a continuously changing pressure wave. How do we store that in our computer?

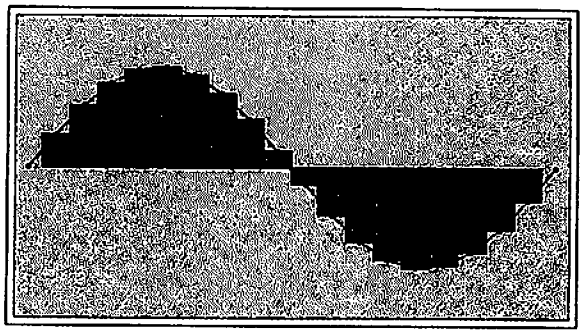

FIGURE 6.9
Area under a curve estimated with rectangles.

By the way, playback systems on computers work essentially the same in reverse. Sound hardware does *digital-to-analog conversion (DAC)*, and the analog signal is then sent to the speakers. The DAC process also requires numbers representing pressure.

If you know calculus, you have some idea of how we might do this. You know that we can get close to measuring the area under a curve with more and more rectangles whose height matches the curve (Figure 6.9). With this idea, it's pretty clear that if we capture enough of those microphone pressure readings, we capture the wave. We call each pressure reading a *sample*—we are literally "sampling" the sound at that moment. But how many samples do we need? In integral calculus, you compute the area under the curve by (conceptually) having an infinite number of rectangles. While computer memories are growing larger and larger all the time, we can't capture an infinite number of samples per sound.

Mathematicians and physicists wondered about these kinds of questions long before there were computers and the answer to how many samples we need was actually computed long ago. The answer depends on the highest *frequency* you want to capture. Let's say that you don't care about any sounds higher than 8000 Hz. The **Nyquist theorem** says that we would need to capture 16,000 samples per second to completely capture and define a wave whose frequency is less than 8000 cycles per second.

Computer Science Idea: Nyquist Theorem

To capture a sound of at most *n* cycles per second, you need to capture 2*n* samples per second.

■

This isn't just a theoretical result. The Nyquist theorem influences applications in our daily life. It turns out that human voices don't typically get over 4000 Hz. That's why our telephone system is designed around capturing 8000 samples per second. That's why playing music through the telephone doesn't really work very well. The limit of (most) human hearing is around 22,000 Hz. If we were to capture 44,000 samples per second, we would be able to capture any sound that we could actually hear. CD's are created by capturing sound at 44,100 samples per second—just a little bit more than 44 kHz for technical reasons and for a fudge factor.

We call the rate at which samples are collected the *sampling rate*. Most sounds that we hear in daily life are at frequencies far below the limits of our hearing. You can capture and manipulate sounds in this class at a sampling rate of 22 kHz (22,000 samples per second) and they will sound quite reasonable. If you use too low a sampling rate to capture a high-pitched sound, you'll still hear something when you play the sound back, but the pitch will sound strange.

Typically, each of these samples are encoded in 2 bytes, or 16 bits. Though there are larger *sample size*, 16 bits works perfectly well for most applications. CD-quality sound uses 16-bit samples.

6.1.4 Binary Numbers and Two's Complement

In 16 bits, the numbers that can be encoded range from −32,768 to 32,767. These aren't magic numbers—they make perfect sense when you understand the encoding. These numbers are encoded in 16 bits using a technique called **two's complement notation** but we can understand it without knowing the details of that technique. We've got 16 bits to represent positive and negative numbers. Let's set aside one of these bits (remember, it's just 0 or 1) to represent whether we're talking about a positive (0) or negative (1) number. We call this the *sign bit*. That leaves 15 bits to represent the actual value. How many different patterns of 15 bits are there? We could start counting:

```
000000000000000
000000000000001
000000000000010
000000000000011
...
111111111111110
111111111111111
```

This looks foreboding. Let's see if we can figure out a pattern. If we've got two bits, there are four patterns: 00, 01, 10, 11. If we've got three bits, there are eight patterns: 000, 001, 010, 011, 100, 101, 110, 111. It turns out that 2^2 is four, and 2^3 is eight. Play with four bits. How many patterns are there? $2^4 = 16$. It turns out that we can state this as a general principle.

Computer Science Idea: 2^n Patterns in *n* Bits

If you have *n* bits, there are 2^n possible patterns in those *n* bits. ■

$2^{15} = 32,768$. Why is there one more value in the negative range than the positive? Zero is neither negative nor positive, but if we want to represent it as bits, we need to define some pattern as zero. We use one of the positive range values (where the sign bit is zero) to represent zero, so that it takes up one of the 32,768 patterns.

The way that computers often represent positive and negative integers is called *two's complement*. In two's complement, positive numbers are shown as usual in binary. The number 9 is 00001001 in binary. The two's complement of a negative number can be calculated by starting with the positive number in binary and inverting it so that all the

```
 1111111
 00001001
+11110111
 00000000
```

FIGURE 6.10
Adding 9 and −9 in two's complement.

1's become 0's and all the 0's become 1's. Finally, add 1 to the result. So −9 starts as 00001001 which after inversion is 11110110 and then adding 1 results in 11110111. One advantage to representing numbers in two's complement is that if you add a negative number (−3) to the positive number of the same value (3), the result is zero, since 1 plus 1 is 0, carry the 1 (Figure 6.10).

6.1.5 Storing Digitized Sounds

The sample size is a limitation on the amplitude of the sound that can be captured. If you have a sound that generates a pressure greater than 32,767 (or a rarefaction greater than −32,768), you'll only capture up to the limits of the 16 bits. If you were to look at the wave in the signal view, it would look like somebody had taken some scissors and *clipped* off the peaks of the waves. We call this effect *clipping* for that very reason. If you play (or generate) a sound that's clipped, it sounds bad—it sounds like your speakers are breaking.

There are other ways of digitizing sound but this is by far the most common. The technical term for this way of encoding sound is *pulse coded modulation (PCM)*. You may encounter this term if you read further in audio or play with audio software.

What this means is that a sound in a computer is a long list of numbers, each of which is a sample in time. There is an ordering in these samples: if you played the samples out of order, you wouldn't get the same sound at all. The most efficient way to store an ordered list of data items on a computer is with an **array**. An array is literally a sequence of bytes right next to one another in memory. We call each value in an array an **element**.

We can easily store the samples that make up a sound in an array. Think of each two bytes as storing a single sample. The array will be large—for CD-quality sounds, there will be 44,100 elements for every second of recording. A minute-long recording will result in an array with 26,460,000 elements.

Each array element has a number associated with it called its **index**. The index numbers increase sequentially. The first element of the array is at index 0, the second one is at index 1, and so on. The last element in the array is at an index equal to the number of elements in the array minus 1. You can think about an array as a long line of boxes, each one holding a value and each box having an index number (Figure 6.11).

Using the MediaTools, you can explore a sound (Figure 6.12) and get a sense of where the sound is quiet (small amplitudes) and loud (large amplitudes). This is important if you want to manipulate the sound. For example, the gaps between recorded words tend to be quiet—at least quieter than the words themselves. You can pick out where words end by looking for the gaps, as in Figure 6.12.

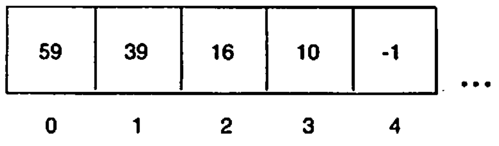

FIGURE 6.11
A depiction of the first five elements in a real sound array.

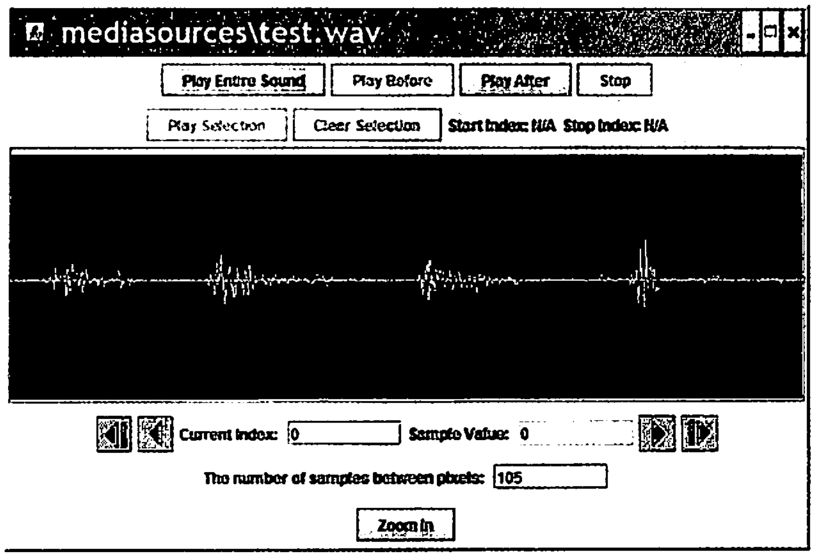

FIGURE 6.12
A sound recording graphed in MediaTools.

You will soon read about how to read a file containing a recording of a sound into a *sound object*, view the samples in the sound, and change the values of the sound array elements. By changing the values in the array, you change the sound. Manipulating a sound is simply a matter of manipulating the elements in an array.

6.2 MANIPULATING SOUNDS

Now that we know how sounds are encoded, we can manipulate sounds using our Python programs. Here's what we'll need to do.

1. We'll need to get the filename of a WAV file and make a sound from it.
2. You will often get the samples of the sound. Sample objects are easy to manipulate, and they know that when you change them, they should automatically change the original sound. You'll read first about manipulating the samples to start with, then about how to manipulate the sound samples from within the sound itself.
3. Whether you get the sample objects out of a sound or just deal with the samples in the sound object, you will then want to do something to the samples.