Introduction to Media Computation: A Multimedia Cookbook in Python

Mark Guzdial

November 16, 2002

Contents

List of Figures

Preface

This book is based on the proposition that very few people actually want to learn to program. However, most educated people *want* to use a computer, and the task that they most want to do with a computer is *communicate.* Alan Perlis first made the claim in 1961 that computer science, and programming explicitly, should be part of a liberal education [Greenberger, 1962]. However, what we've learned since then is that one doesn't just "learn to program." One learns to program *something* [Adelson and Soloway, 1985, Harel and Papert, 1990], and the motivation to do that something can make the difference between learning to program or not [Bruckman, 2000].

The philosophies which drive the structure of this book include:

- People learn concrete to abstract, driven by need. Teaching structure before content is painful and results in brittle knowledge that can't be used elsewhere [Bruer, 1993]. Certainly, one can *introduce* structure (and theory and design), but students won't really understand the structure until they have the content to fill it with – and a reason to need the structure. Thus, this book doesn't introduce debugging or design (or complexity or most of computer science) until the students are doing complex enough software to make it worthwhile learning.

- Repetition is good. Variety is good. Marvin Minsky once said, "If you know something only one way, you don't know it at all." The same ideas come back frequently in this book. The same idea is framed in multiple ways. I will use metaphor, visualizations, mathematics, and even computer science to express ideas in enough different ways that *one* of the ways will ring true for the individual student.

- The computer is the most amazingly creative device that humans have ever conceived of. It is literally completely made up of mind-stuff. As the movie says, "Don't just dream it, be it." If you can imagine it, you can make it “real” on the computer. Playing with programming can be and *should* be enormous fun.

Typographical notations

Examples of Python code look like this: `x = x + 1`. Longer examples look look like this:

```
def helloWorld():
  print "Hello, world!"
```

When showing something that the user types in with Python’s response, it will have a similar font and style, but the user’s typing will appear after a Python prompt (>>>):

```
>>> print 3 + 4
7
```

User interface components of JES (Jython Environment for Students) will be specified using a smallcaps font, like save menu item and the Load button.

There are several special kinds of sidebars that you’ll find in the book.

Recipe 1: An Example Recipe

Recipes (programs) appear like this:

```
def helloWorld():
  print "Hello, world!"
```

Computer Science Idea: An Example Idea
Key computer science concepts appear like this.

Common Bug: An Example Common Bug
Common things that can cause your recipe to fail appear like this.

Debugging Tip: An Example Debugging Tip
If there's a good way to keep those bugs from creeping into your recipes in the first place, they're highlighted here.

Making it Work Tip: An Example How To Make It Work
Best practices or techniques that really help are highlighted like this.

Acknowledgements

Our sincere thanks go out to ···

- Jason Ergle, Claire Bailey, David Raines, and Joshua Sklare who made JES a reality with amazing quality for such a short amount of time. Jason and David took JES the next steps, improving installation, debugging, and process support.

- Adam Wilson built the MediaTools that are so useful for exploring sounds and images and processing video.

- Andrea Forte, Mark Richman, Matt Wallace, Alisa Bandlow, and David Rennie who helped build the course materials. Mark and Matt built a great many of the example programs.

- Jeff Pierce for reviewing and advising us on the design of the media language used in the book.

- Bob McMath, Vice-Provost at Georgia Tech, and Jim Foley, Associate Dean for Education in the College of Computing, for investing in this effort early on.

- Kurt Eiselt who worked hard to make this effort real, convincing others to take it seriously.

- Janet Kolodner and Aaron Bobick who were excited and encouraging about the idea of media computation for students new to computer science.

- Joan Morton, Chrissy Hendricks, and all the staff of the GVU Center who made sure that we had what we needed and that the paperwork was done to make this class come together.

- Picture of Alan Perlis from `http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/csd/perlis.html`. Picture of Alan Turing from `http://www.dromo.com/fusionanomaly/applecomputer.html`. Picture of Grace Hopper from `http://www.autodesk.com/dyf/ginvolved/december.html`. Most of the clip art is used with permission from the *Art Explosion* package by Nova Development.

- Finally but most importantly, Barbara Ericson, and Matthew, Katherine, and Jennifer Guzdial, who allowed themselves to be photographed and recorded for Daddy's media project and were supportive and excited about the class.

Part I

Introduction

Chapter 1

Introduction to Media Computation

1.1 What is computer science about?

Computer science is the study of *process*: How we do things, how we specify what we do, how we specify what the stuff is that you're processing. But that's a pretty dry definition. Let's try a metaphorical one.

Computer Science Idea: Computer science is the study of recipes

They're a special kind of recipe—one that can be executed by a computational device, but that point is only of importance to computer scientists. The important point overall is that a computer science recipe defines *exactly* what's to be done.

If you're a biologist who wants to describe how migration works or how DNA replicates, or if you're a chemist who wants to explain how an equilibrium is reached in a reaction, or if you're a factory manager who wants to define a machine-and-belt layout and even test how it works before physically moving heavy things into position, then being able to write a recipe that specifies *exactly* what happens, in terms that can be completely defined and understood, is *very* useful. This exactness is part of why computers have radically changed so much of how science is done and understood.

It may sound funny to call *programs* or *algorithms* a recipe, but the analogy goes a long way. Much of what computer scientists study can be defined in terms of recipes:

- Some computer scientists study how recipes are written: Are there better or worse ways of doing something? If you've ever had to separate whites from yolks in eggs, you know that knowing the right way to do it makes a world of difference. Computer science theoreticians worry about the fastest and shortest recipes, and the ones that take up the least amount of space (you can think about it as counter space — the analogy works). *How* a recipe works, completely apart from how it's written, is called the study of *algorithms*. Software engineers worry about how large groups can put together recipes that still work. (The recipe for some programs, like the one that keeps track of Visa/MasterCard records has literally millions of steps!)

- Other computer scientists study the units used in recipes. Does it matter whether a recipe uses metric or English measurements? The recipe may work in either case, but if you have the read the recipe and you don't know what a pound or a cup is, the recipe is a lot less understandable to you. There are also units that make sense for some tasks and not others, but if you can fit the units to the tasks well, you can explain yourself more easily and get things done faster—and avoid errors. Ever wonder why ships at sea measure their speed in *knots*? Why not use things like meters per second? There are places, like at sea, where more common terms aren't appropriate or don't work as well. The study of computer science units is referred to as *data structures*. Computer scientists who study ways of keeping track of lots of data in lots of different kinds of units are studying *databases*.

- Can recipes be written for anything? Are there some recipes that *can't* be written? Computer scientists actually do know that there are recipes that can't be written. For example, you can't write a recipe that can absolutely tell, for any other recipe, if the other recipe will actually work. How about *intelligence*? Can we write a recipe that can *think* (and how would you tell if you got it right)? Computer scientsts in *theory*, *intelligent systems*, *artificial intelligence*, and *systems* worry about things like this.

- There are even computer scientists who worry about whether people like what the recipes produce, like the restauraunt critics for the newspaper. Some of these are *human-computer interface* specialists who worry about whether people like how the recipes work (those "recipes" that produce an *interface* that people use, like windows, buttons, scrollbars, and other elements of what we think about as a running program).

- Just as some chefs specialize in certain kinds of recipes, like crepes or barbeque, computer scientists also specialize in special kinds of recipes. Computer scientists who work in *graphics* are mostly concerned with recipes that produce pictures, animations, and even movies. Computer scientists who work in *computer music* are mostly concerned with recipes that produce sounds (often melodic ones, but not always).

- Still other computer scientists study the *emergent properties* of recipes. Think about the World Wide Web. It's really a collection of *millions* of recipes (programs) talking to one another. Why would one section of the Web get slower at some point? It's a phenomena that emerges from these millions of programs, certainly not something that was planned. That's something that *networking* computer scientists study. What's really amazing is that these emergent properties (that things just start to happen when you have many, many recipes interacting at once) can also be used to explain non-computational things. For example, how ants forage for food or how termites make mounds can also be described as something that just happens when you have lots of little programs doing something simple and interacting.

The recipe metaphor also works on another level. Everyone knows that some things in recipe can be changed without changing the result dramatically. You can always increase all the units by a multiplier to make more. You can always add more garlic or oregano to the spaghetti sauce. But there are some things that you cannot change in a recipe. If the recipe calls for baking powder, you may not substitute baking soda. If you're supposed to boil the dumplings then saute' them, the reverse order will probably not work well.

Similarly, for software recipes. There are usually things you can easily change: The actual names of things (though you should change names consistently), some of the *constants* (numbers that appear as plain old numbers, not as variables), and maybe even some of the data *ranges* (sections of the data) being manipulated. But the order of the commands to the computer, however, almost always has to stay exactly as stated. As we go on, you'll learn what can be changed safely, and what can't.

Computer scientists specify their recipes with *programming languages*. Different programming languages are used for different purposes. Some of them are wildly popular, like Java and C++. Others are more obscure, like Squeak and T. Others are designed to make computer science ideas very easy to learn, like Scheme or Python, but the fact that they're easy to learn doesn't always make them very popular nor the best choice for experts building larger or more complicated recipes. It's a hard balance in teaching computer science to pick a language that is easy to learn *and* is popular and useful enough that students are motivated to learn it.

Why don't computer scientists just use natural languages, like English or Spanish? The problem is that natural languages evolved the way that they did to enhance communications between very smart beings, humans. As we'll go into more in the next section, computers are exceptionally dumb. They need a level of specificity that natural language isn't good at. Further, what we say to one another in natural communication is not exactly what you're saying in a computational recipe. When was the last time you told someone how a videogame like Doom or Quake or Super Mario Brothers worked in such minute detail that they could actually replicate the game (say, on paper)? English isn't good for that kind of task.

There are so many different kinds of programming languages because there are so many different kinds of recipes to write. Programs written in the programming language *C* tend to be very fast and efficient, but they also tend to be hard to read, hard to write, and require units that are more about computers than about bird migrations or DNA or whatever else you want to write your recipe about. The programming language *Lisp* (and its related languages like Scheme, T, and Common Lisp) is very flexible and is well suited to exploring how to write recipes that have never been written before, but Lisp *looks* so strange compared to languages like C that many people avoid it and there are (natural consequence) few people who know it. If you want to hire a hundred programmers to work on your project, you're going to find it easier to find a hundred programmers who know a popular language than a less popular one—but that doesn't mean that the popular language is the best one for your task!

The programming language that we're using in this book is *Python* (`http://www.python.org` for more information on Python). Python is a fairly popular programming language, used very often for Web and media programming. The web search engine *Google* is mostly programmed in Python. The media company *Industrial Light & Magic* also uses Python. Python is known for being easy to learn, easy to read, very flexible, but not very efficient. The same algorithm coded in C and in Python will probably be faster in C. The version of Python that we're using is called *Jython* (`http://www.jython.org`). Python is normally implemented in the programming language C. Jython is Python implemented in Java. Jython lets us do multimedia that will work across multiple computer platforms.

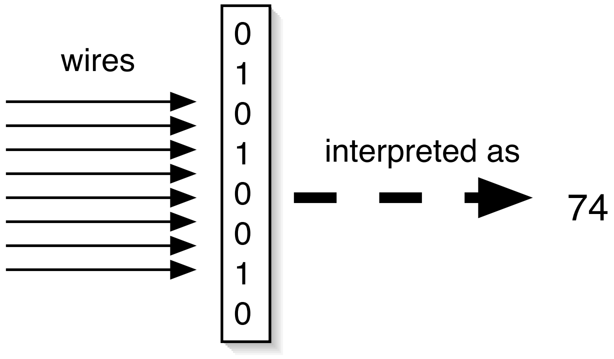


Figure 1.1: Eight wires with a pattern of voltages is a byte, which gets interpreted as a pattern of eight 0's and 1's, which gets interpreted as a decimal number.

1.2 What Computers Understand

Computational recipes are written to run on computers. What does a computer know how to do? What can we tell the computer to do in the recipe? The answer is "Very, very little." Computers are exceedingly stupid. They really only know about numbers.

Actually, even to say that computers know numbers is a myth, or more appropriately, an *encoding*. Computers are electronic devices that react to voltages on wires. We group these wires into sets (like eight of these wires are called a *byte* and one of them is called a *bit*). If a wire has a voltage on it, we say that it encodes a 1. If it has no voltage on it, we say that it encodes a 0. So, from a set of eight wires (a byte), we interpret a pattern of eight 0's and 1's, e.g., 01001010. Using the *binary* number system, we can interpret this byte as a *decimal number* (Figure 1.1). That's where we come up with the claim that a computer knows about numbers[1].

The computer has a *memory* filled with bytes. Everything that a computer is working with at a given instant is stored in its memory. That means that everything that a computer is working with is *encoded* in its bytes: JPEG pictures, Excel spreadsheets, Word documents, annoying Web pop-up ads, and the latest spam email.

A computer can do lots of things with numbers. It can add them, subtract them, multiply them, divide them, sort them, collect them, duplicate

[1] We'll talk more about this level of the computer in Chapter 13

them, filter them (e.g., "make a copy of these numbers, but only the even ones"), and compare them and do things based on the comparison. For example, a computer can be told in a recipe "Compare these two numbers. If the first one is less than the second one, jump to step 5 in this recipe. Otherwise, continue on to the next step."

So far, the computer is an incredible calculator, and that's certainly why it was invented. The first use of the computer was during World War II for calculating trajectories of projectiles ("If the wind is coming from the SE at 15 MPH, and you want to hit a target 0.5 miles away at an angle of 30 degrees East of North, then incline your launcher to ..."). The computer is an amazing calculator. But what makes it useful for general recipes is the concept of *encodings.*

Computer Science Idea: Computers can layer encodings
Computers can layer encodings to virtually any level of complexity. Numbers can be interpreted as characters, which can be interpreted in sets as Web pages, which can be interpreted to appear as multiple fonts and styles. But at the bottommost level, the computer *only* "knows" voltages which we intepret as numbers.

If one of these bytes is interpreted as the number 65, it could just be the number 65. Or it could be the letter *A* using a standard encoding of numbers-to-letters called the *American Standard Code for Information Interchange (ASCII).* If that 65 appears in a collection of other numbers that we're interpreting as text, and that's in a file that ends in ".html" it might be part of something that looks like this `<a href=`..., which a Web browser will interpret as the definition of a link. Down at the level of the computer, that *A* is just a pattern of voltages. Many layers of recipes up, at the level of a Web browser, it defines something that you can click on to get more information.

If the computer understands only numbers (and that's a stretch already), how does it manipulate these encodings? Sure, it knows how to compare numbers, but how does that extend to being able to alphabetize a class list/ Typically, each layer of encoding is implemented as a piece or layer of software. There's software that understands how to manipulate characters. The character software knows how to do things like compare names because it has encoded that *a* comes before *b* and so on, and that the numeric comparison of the order of numbers in the encoding of the letters leads to alphabetical comparisons. The character software is used by other software that manipulates text in files. That's the layer that something like Microsoft Word or Notepad or TextEdit would use. Still another piece of software knows how to interpret *HTML* (the language of the Web), and another layer of that software knows how to take HTML and display the right text, fonts, styles, and colors.

We can similarly create layers of encodings in the computer for our specific tasks. We can teach a computer that cells contain mitochondria and DNA, and that DNA has four kinds of nucleotides, and that factories have these kinds of presses and these kinds of stamps. Creating layers of encoding and interpretation so that the computer is working with the right units (recall back to our recipe analogy) for a given problem is the task of *data representation* or defining the right *data structures*.

If this sounds like a lot of software, it is. When software is layered like this, it slows the computer down some. But the amazing thing about computers is that they're *amazingly* fast—and getting faster all the time!

Computer Science Idea: Moore's Law
Gordon Moore, one of the founders of Intel (maker of computer processing chips for all computers running Windows operating systems), made the claim that the number of transistors (a key component of computers) would double at the same price every 18 months, effectively meaning that the same amount of money would buy twice as much computing power every 18 months. That means, in a year-and-a-half, computers get as fast over again as has taken them since World War II. This Law has continued to hold true for decades.

Computers today can execute literally *BILLIONS* of recipe steps per second! They can hold in memory literally encyclopediae of data! They never get tired nor bored. Search a million customers for a particular card holder? No problem! Find the right set of numbers to get the best value out of an equation? Piece of cake!

Process millions of picture elements or sound fragments or movie frames? That's *media computation*.

1.3 Media Computation: Why digitize media?

Let's consider an encoding that would be appropriate for pictures. Imagine that pictures were made up of little dots. That's not hard to imagine: Look really closely at your monitor or at a TV screen and see that your images are *already* made up of little dots. Each of these dots is a distinct color. You know from your physics that colors can be described as the sum of *red*, *green*, and *blue*. Add the red and green to get yellow. Mix all three together to get white. Turn them all off, and you get a black dot.

What if we encoded each dot in a picture as collection of three bytes, one each for the amount of red, green, and blue at that dot on the screen? And we collect a bunch of these three-byte-sets to determine all the dots of a given picture? That's a pretty reasonable way of representing pictures, and it's essentially how we're going to do it in Chapter 5.

Manipulating these dots (each referred to as a *pixel* or *picture element*) can take a lot of processing. There are thousands or even millions of them in a picture that you might want to work with on your computer or on the Web. But the computer doesn't get bored and it's mighty fast.

The encoding that we will be using for sound involves 44,100 two-byte-sets (called a *sample*) for each *second* of time. A three minute song requires 158,760,000 bytes. Doing any processing on this takes a *lot* of operations. But at a billion operations per second, you can do lots of operations to every one of those bytes in just a few moments.

Creating these kinds of encodings for media requires a change to the media. Look at the real world: It isn't made up of lots of little dots that you can see. Listen to a sound: Do you hear thousands of little bits of sound per second? The fact that you *can't* hear little bits of sound per second is what makes it possible to create these encodings. Our eyes and ears are limited: We can only perceive so much, and only things that are just so small. If you break up an image into small enough dots, your eyes can't tell that it's not a continuous flow of color. If you break up a sound into small enough pieces, your ears can't tell that the sound isn't a continuous flow of auditory energy.

The process of encoding media into little bits is called *digitization*, sometimes referred to as "*going digital*." *Digital* means (according to the American Heritage Dictionary) "Of, relating to, or resembling a digit, especially a finger." Making things digital is about turning things from continuous, uncountable, to something that we can count, as if with our fingers.

Digital media, done well, feel the same to our limited human sensory apparatus as the original. Phonograph recordings (ever seen one of those?)

capture sound continuously, as an *analogue* signal. Photographs capture light as a continuous flow. Some people say that they can hear a difference between phonograph recordings and CD recordings, but to my ear and most measurements, a CD (which *is* digitized sound) sounds just the same—maybe clearer. Digital cameras at high enough resolutions produce photograph-quality pictures.

Why would you want to digitize media? Because it's easier to manipulate, to replicate exactly, to compress, and to transmit. For example, it's hard to manipulate images that are in photographs, but it's very easy when the same images are digitized. This book is about using the increasingly digital world of media and manipulating it—and learning computation in the process.

Moore's Law has made media computation feasible as an introductory topic. Media computation relies on the computer doing lots and lots of operations on lots and lots of bytes. Modern computers can do this easily. Even with slow (but easy to understand) languages, even with inefficient (but easy to read and write) recipes, we can learn about computation by manipulating media.

1.4 Computer Science for Non-Computer Scientists

But why should you? Why should anyone who doesn't want to be a computer scientist learn about computer science? Why should you be interested in learning about computation through manipulating media?

Most professionals today do manipulate media: Papers, videos, tape recordings, photographs, drawings. Increasingly, this manipulation is done with a computer. Media are very often in a digitized form today.

We use software to manipulate these media. We use Adobe Photoshop for manipulating our images, and Macromedia SoundEdit to manipulate our sounds, and perhaps Microsoft PowerPoint for assembling our media into slideshows. We use Microsoft Word for manipulating our text, and Netscape Navigator or Microsoft Internet Explorer for browsing media on the Internet.

So why should anyone who does *not* want to be a computer scientist study computer science? Why should you learn to program? Isn't it enough to learn to *use* all this great software? The following two sections provide two answers to these questions.

1.4.1 It's about communication

Digital media are manipulated with software. *If you can only manipulate media with software that* ***someone else*** *made for you, you are limiting your ability to communicate.* What if you want to say something or say it in some way that Adobe, Microsoft, Apple, and the rest don't support you in saying? If you know how to program, even if it would take you *longer* to do it yourself, you have that freedom.

What about learning those tools in the first place? In my years in computers, I've seen a variety of software come and go as *the* package for drawing, painting, word-processing, video editing, and beyond. You can't learn just a single tool and expect to be able to use that your entire career. If you know *how* the tools work, you have a core understanding that can transfer from tool to tool. You can think about your media work in terms of the *algorithms*, not the *tools*.

Finally, if you're going to prepare media for the Web, for marketing, for print, for broadcast, for any use whatsoever, it's worthwhile for you to have a sense of what's possible, what can be done with media. It's even more important as a consumer of media that you know how the media can be manipulated, to know what's true and what could be just a trick. If you know the basics of media computation, you have an understanding that goes beyond what any individual tool provides.

1.4.2 It's about process

In 1961, Alan Perlis gave a talk at MIT where he made the argument that computer science, and programming explicitly, should be part of a general, liberal education [Greenberger, 1962]. Perlis is an important figure in the field of computer science (Figure 1.2). The highest award that a computer scientist can be honored with is the ACM Turing Award. Perlis was the first recipient of that award. He's an important figure in software engineering, and he started several of the first computer science departments in the United States.

Perlis' argument was made in comparison with calculus. Calculus is generally considered part of a liberal education: Not *everyone* takes calculus, but if you want to be well-educated, you will typically take at least a term of calculus. Calculus is the study of *rates*, which is important in many fields. Computer science, as we said before (page 7), is the study of *process*. Process is important to nearly every field, from business to science to medicine to law. Knowing process formally is important to everyone.

Figure 1.2: Alan Perlis

Exercises

Exercise 1: Find an ASCII table on the Web: A table listing every character and its corresponding numeric representation.

Exercise 2: Find a *Unicode* table on the Web. What's the difference between ASCII and Unicode?

Exercise 3: Consider the representation for pictures described in Section 1.3, where each "dot" (pixel) in the picture is represented by three bytes, for the red, green, and blue components of the color at that dot. How many bytes does it take to represent a 640x480 picture, a common picture size on the Web? How many bytes does it take to represent a 1024x768 picture, a common screen size? (What do you think is meant now by a "3 megapixel" camera?)

Exercise 4: How many different numbers can be represented by one byte? In other words, eight bits can represent from zero to what number? What if you have two bytes? Four bytes?

Exercise 5: (Hard) How might you represent a *floating point number* in terms of bytes?

Exercise 6: Look up Alan Kay and the *Dynabook* on the Web. Who is he, and what does he have to do with media computation?

Exercise 7: Look up Alan Turing on the Web. Who was he, and what does he have to do with our notion of what a computer can do and how encodings work?

Exercise 8: Look up Kurt Goedel on the Web. Who was he, and what amazing things did he do with encodings?

To Dig Deeper

James Gleick's book *Chaos* describes more on emergent properties.

Mitchel Resnick's book *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds* [Resnick, 1997] describes how ants, termites, and even traffic jams and slime molds can be described pretty accurately with hundreds or thousands of very small programs running and interacting all at once.

Beyond the Digital Domain [Abernethy and Allen, 1998] is a wonderful introductory book to computation with lots of good information about digital media.

Chapter 2

Introduction to Programming

2.1 Programming is about Naming

Computer Science Idea: Much of programming is about naming

A computer can associate names, or *symbols*, with just about anything: With a particular byte; with a collection of bytes making up a numeric variable or a bunch of letters; with a media element like a file, sound, or picture; or even with more abstract concepts, like a named recipe (a *program*) or a named encoding (a *type*). A computer scientist sees a choice of names as being high quality in the same way that a philosopher or mathematician might: If the naming scheme (the names and what they name) are elegant, parsimonious, and usable.

Obviously, the computer itself doesn't *care* about names. If the computer is just a calculator, then remembering words and the words' association with values is just a waste of the computer's memory. But for humans, it's *very* powerful. It allows us to work with the computer in a natural way, even a way that extends how we think about recipes (processes) entirely.

XXX This section needs work

A *programming language* is really a set of names that a computer has encodings for, such that those names make the computer do expected actions and interpret our data in expected ways. Some of the programming language's names allow us to define *new* namings—which allows us to create our own layers of encoding. Assigning a variable to a value is one way of defining a name for the computer. Defining a function is giving a name to a recipe.

There are good names and less good names. That has nothing to do with curse words, nor with TLA's (Three Letter Acronyms). A good set of encodings and names allow one to describe recipes in a way that's natural, without having to say too much. The variety of different programming languages can be thought of as a collection of sets of namings-and-encodings. Some are better for some tasks than others. Some languages require you to write more to describe the same recipe than others—but sometimes that "more" leads to a much more (human) readable recipe that helps others to understand what you're saying.

Philosophers and mathematicians look for very similar senses of quality. They try to describe the world in few words, but an elegant selection of words that cover many situations, while remaining understandable to their fellow philosophers and mathematicians. That's exactly what computer scientists do as well.

How the units and values (*data*) of a recipe can be interpreted is often also named. Remember how we said in Section 1.2 (page 11) that everything is in bytes, but we can interpret those bytes as numbers? In some programming languages, you can say explicitly that some value is a *byte*, and later tell the language to treat it as a number, an *integer* (or sometimes *int*). Similarly, you can tell the computer that these series of bytes is a collection of numbers (an *array of integers*), or a collection of characters (a *string*), or even as a more complex encoding of a single floating point number (a *float*—any number with a decimal point in it).

In Python, we will explicitly tell the computer how to interpret our values, but we will only rarely tell the computer that certain names only are associated with certain encodings. Languages such as Java and C++ are *strongly typed*: Their names are strongly associated with certain types or encodings.They require you to say that this name will only be associated with integers, and that one will only be a floating point number. Python still has *types* (encodings that you can reference by name), but they're not as explicit.

2.1.1 Files and their Names

A programming language isn't the only place where computers associate names and values. Your computer's *operating system* takes care of the files on your disk, and it associates names with those files. Operating systems you may be familiar with or use include Windows 95, Windows 98 (Windows ME, NT, XP...), MacOS, and Linux. A *file* is a collection of values (bytes) on your *hard disk* (the part of your computer that stores things after the power gets turned off). If you know the name of a file, you can tell it to the operating system, and it can give you the values associated with that name.

You may be thinking, "I've been using the computer for years, and I've *never* 'given a file name to the operating system.' " Maybe not explicitly, but when you pick a file from a file choosing dialog in Photoshop, or double-click a file in a *directory* window (or Explorer or Finder), you are asking some software somewhere to give the name you're picking or double-clicking to the operating system, and get the values back. When you write your own recipes, though, you'll be explicitly getting file names and asking for their values.

Files are *very* important for media computation. Disks can store acres and acres of information on them. Remember our discussion of Moore's Law (page 13)? Disk capacity per dollar is increasing *faster* than computer speed per dollar! Computer disks today can store whole movies, tons of sounds, and tons of pictures.

These media are not small. Even in a *compressed* form, screen size pictures can be over a million bytes large, and songs can be three million bytes or more. You need to keep them someplace where they'll last past the computer being turned off and where there's lots of space.

In contrast, your computer's *memory* is impermanent (disappears when the power does) and is relatively small. Computer memory is getting larger all the time, but it's still just a fraction of the amount of space on your disk. When you're working with media, you will load the media from the disk into memory, but you wouldn't want it to stay in memory after you're done. It's too big.

Think about your computer's memory as a dorm room. You can get to things easily in a dorm room—they're right at hand, easy to reach, easy to use. But you wouldn't want to put everything you own (or everything you hope to own) in that one dorm room. All your belongings? Your skis? Your car? Your boat? That's silly. Instead, you store large things in places designed to store large things. You know how to get them when you need them (and maybe take them back to your dorm room if you need to or can).

When you bring things into memory, you will name the value, so that you can retrieve it and use it later. In that sense, programming is something like *algebra*. To write generalizable equations and functions (those that work for any number or value), you wrote equations and functions with *variables*, like $PV = nRT$ or $e = Mc^2$ or $f(x) = sin(x)$. Those P's, V's, R's, T's, e's, M's, c's, and x's were names for values. When you evaluated $f(30)$, you knew that the x was the name for 30 when computing f. We'll be naming media (as values) in the same way when using them when programming.

2.2 Programming in Python

As we said in the last chapter, the programming language that we're going to be using in this class is called *Python*. It's a language invented by Guido van Rossum. van Rossum named his language for the famous British comedy troupe *Monty Python*. Python has been used for years by people without formal computer science training—it's aimed at being easy to use. The particular form of Python that we're going to be using is *Jython* because it lends itself to cross-platform multimedia.

You'll actually be programming using a tool called *JES* for "*Jython Environment for Students*." JES is a simple *editor* (tool for entering program text) and interaction space so that you can try things out in JES and create new recipes within it. The media names (functions, variables, encodings) that we'll be talking about in this book were developed to work from within JES (i.e., they're not part of a normal Jython distribution, though the basic language we'll be using is normal Python).

To install JES, you'll have to do these things:

1. Make sure that you have Java installed on your computer. If you don't have it, you can get it from the Sun site at `http://www.java.sun.com`.

2. You'll need to install Jython. You will probably have a CD accompanying this text including both Jython and JES. You will just unzip the archive to get Jython and JES set up.

3. If you don't have a CD, you'll need the individual components. You'll be able to get Jython from `http://www.jython.org` and JES from `http://coweb.cc.gatech.edu/mediaComp-plan/Gettingsetup`. Be sure to unzip JES as a directory *inside* the Jython directory.

2.2.1 Programming in JES

How you start JES depends on your platform. In Linux, you'll probably `cd` into your Jython directory and type a command like `jes`. In Windows or Macintosh, you'll have a `jes.bat` batch file or a `Jes` applet that you can double click from within the Jython directory.

XXX Insert Pictures of What this Looks Like Here

Once you start JES, it will look something like Figure 2.1. (A Windows screenshot of the same thing is Figure 2.2—it really does run the same thing on all Java-supported platforms.) There are two areas in JES (the bar between them moves so that you can differentially resize the two areas):

- The top part is the *program area*. This where you write *your* recipes: The programs and their names that you're creating. This area is simply a text editor—think of it as Microsoft Word for your programs. The computer doesn't actually try to intepret the names that you type up in the program area until you press the LOAD, and you can't press the LOAD button until you've saved your program (by using the SAVE menu item, which is visible in Figure 2.2 under the FILE menu.

- The bottom part is the *command area*. This is where you literally *command* the computer to do something. You type your commands at the `>>>` prompt, and when you hit return, the computer will interpret your words (i.e., apply the meanings and encodings of the Python programming language) and do what you have told it to do. This interpretation will include whatever you typed and loaded from the program area as well.

2.2.2 Media Computation in JES

We're going to start out by simply typing commands in the command area—not defining new names yet, but simply using the names that the computer already knows from within JES.

The name `print` is an important one to know. It's always used with something following it. The meaning for `print` is "Display a readable representation of whatever follows." Whatever follows can be a name that the computer knows, or an *expression* (literally in the algebraic sense). Try typing `print 34 + 56` by clicking in the command area, typing the command, and hitting return—like this:

```
>>> print 34 + 56
90
```

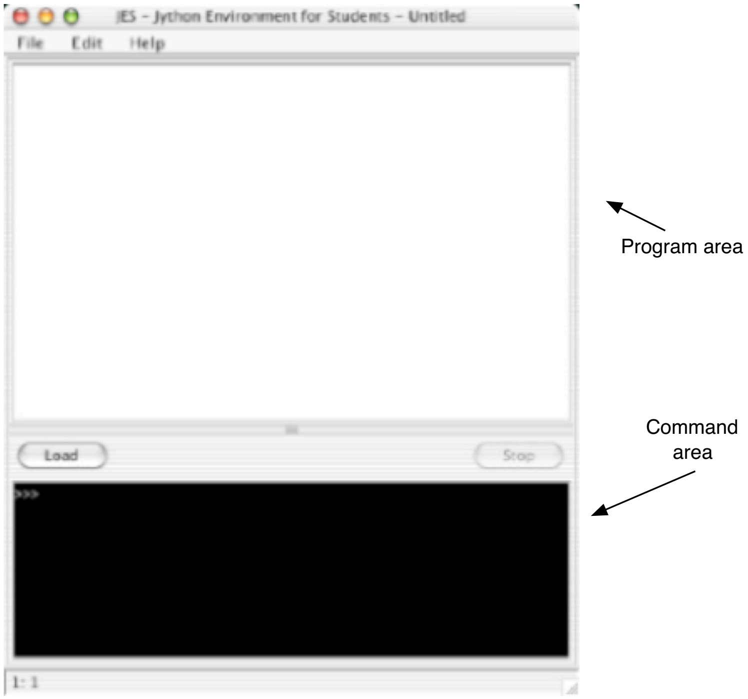
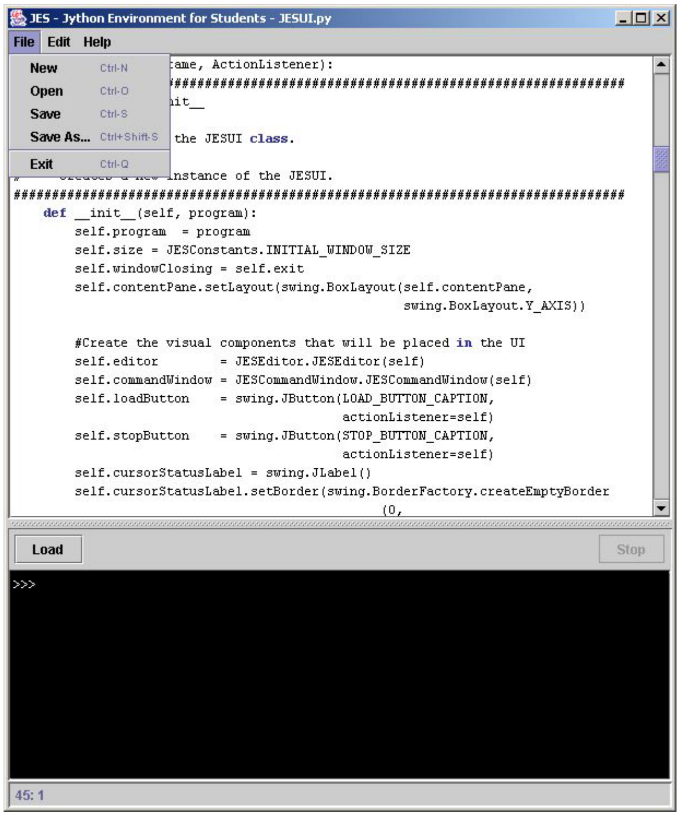


Figure 2.1: JES running on MacOS X (with annotations)

`34 + 56` is a numeric expression that Python understands. Obviously, it's composed of two numbers and an operation (in our sense, a *name*) that Python knows how to do, `+` meaning "add." Python understands other kinds of expressions, not all numeric.

```
>>> print 34.1/46.5
0.7333333333333334
>>> print 22 * 33
726
>>> print 14 - 15
-1
>>> print "Hello"
Hello
>>> print "Hello" + "Mark"
HelloMark
```



```
                   ame, ActionListener):
                   #################################################################
                   it__

                     the JESUI class.

                   instance of the JESUI.
#################################################################################
    def __init__(self, program):
        self.program  = program
        self.size = JESConstants.INITIAL_WINDOW_SIZE
        self.windowClosing = self.exit
        self.contentPane.setLayout(swing.BoxLayout(self.contentPane,
                                                    swing.BoxLayout.Y_AXIS))

        #Create the visual components that will be placed in the UI
        self.editor        = JESEditor.JESEditor(self)
        self.commandWindow = JESCommandWindow.JESCommandWindow(self)
        self.loadButton    = swing.JButton(LOAD_BUTTON_CAPTION,
                                            actionListener=self)
        self.stopButton    = swing.JButton(STOP_BUTTON_CAPTION,
                                            actionListener=self)
        self.cursorStatusLabel = swing.JLabel()
        self.cursorStatusLabel.setBorder(swing.BorderFactory.createEmptyBorder
                                                   (0,
```

```
>>>
```

Figure 2.2: JES running on Windows

Python understands a bunch of standard math operations. It also knows how to recognize different kinds of numbers, both integer and floating point. It also knows how to recognize *strings* (lists of characters) that are started and ended with " (quote) marks. It even knows what it means to “add” two strings together: It simply puts one right after the other.

Common Bug: Python's types can produce odd results

Python takes types seriously. If it sees you using integers, it thinks you want an integer result from your expressions. If it sees you use floating point numbers, it thinks you want a floating point result. Sounds reasonable, no? But how about:

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

`1/2` is `0`? Well, sure! 1 and 2 are integers. There is no integer equal to 1/2, so the answer must be 0! Simply by adding ".0" to an integer convinces Python that we're talking about floating point numbers, so the result is in floating point form..

Python also understands about *functions*. Remember functions from algebra? They're a "box" into which you put one value, and out comes another. One of the functions that Python knows takes a character as the *input* value and returns the number that is the ASCII mapping for that character. The name of that function is `ord` (for *ordinal*), and you can use `print` to display the value that the function `ord` returns:

```
>>> print ord("A")
65
```

Another function that's built in to Python is named `abs`—it's the absolute value function. It returns the absolute value of the input value.

```
>>> print abs(1)
1
>>> print abs(-1)
1
```

Debugging Tip: Common typos

If you type something that Python can't understand at all, you'll get a syntax error.

```
>>> pint "Hello"
A syntax error is contained in the code
-- I can't read it as Python.
```

If you try to access a word that Python doesn't know, Python will say that it doesn't know that name.

```
>>> print a
A local or global name could not be
found.
```

Another function that JES[1] knows is one that allows you to pick a file from your disk. It takes no input, like `ord` did, but it does return a string which is the name of the file on your disk. The name of the function is `pickAFile`. Python is very picky about capitalization—neither `pickafile` nor `Pickafile` will work! Try it like this `print pickAFile()`. When you do, you will get something that looks like Figure 2.3.

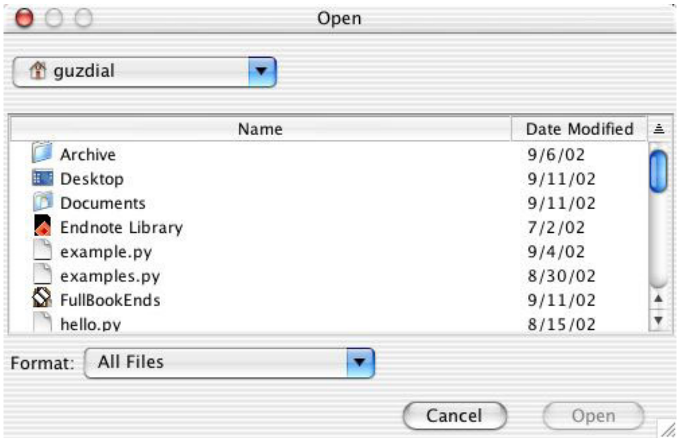
You're probably already familiar with how to use a file picker or file dialog like this:

- Double-click on folders/directories to open them.
- Click to select and then click Open, or double-click, to select a file.

Once you select a file, what gets returned is the *file name* as a string (a sequence of characters). (If you click Cancel, `pickAFile` returns the *empty string*—a string of characters, with no characters in it, e.g., `""`.) Try it: Do `print pickAFile()` and Open a file.

```
>>> print pickAFile()
/Users/guzdial/mediasources/barbara.jpg
```

[1] You might notice that I switched from saying "Python" knows to "JES" knows. `print` is something that all Python implementations know. `pickAFile` is something that we built for JES. In general, you can ignore the difference, but if you try to use another kind of Python, it'll be important to know what's common and what isn't.



Name	Date Modified
Archive	9/6/02
Desktop	9/11/02
Documents	9/11/02
Endnote Library	7/2/02
example.py	9/4/02
examples.py	8/30/02
FullBookEnds	9/11/02
hello.py	8/15/02

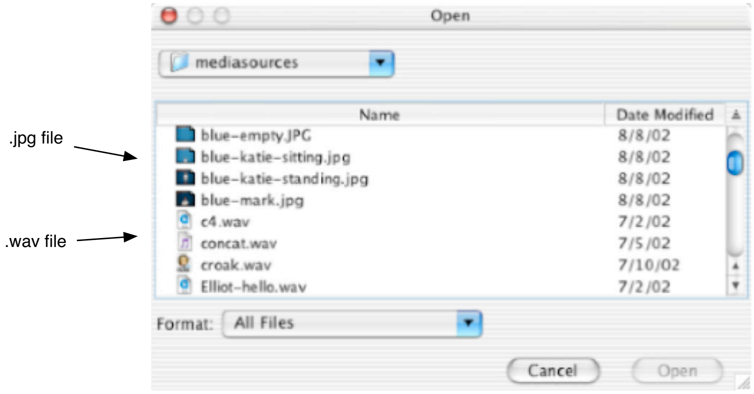
Figure 2.3: The File Picker

What *you* get when you finally select a file will depend on your operating system. On Windows, your file name will probably start with `C:` and will have backslashes in it (e.g., \). On Linux or MacOS, it will probably look something like the above. There are really two parts to this file name:

- The character between words (e.g., the `/` between "Users" and "guzdial") is called the *path delimiter*. Everything from the beginning of the file name to the last path delimiter is called the *path* to the file. That describes exactly *where* on the hard disk (in which *directory*) a file exists.

- The last part of the file (e.g. "barbara.jpg") is called the *base file name*. When you look at the file in the Finder/Explorer/Directory window (depending on your operating system), that's the part that you see. Those last three characters (after the period) is called the *file extension*. It identifies the encoding of the file.

Files that have an extension of ".jpg" are *JPEG* files. They contain pictures. JPEG is a standard encoding for any kind of images. The other kind of media files that we'll be using alot are ".wav" files (Figure 2.4). The ".wav" extension means that these are *WAV* files. They contain sounds. WAV is a standard encoding for sounds. There are many other kinds of extensions for files, and there are even many other kinds of media extensions. For example, there are also GIF (".gif") files for images and AIFF (".aif" or ".aiff") files for sounds. We'll stick to JPEG and WAV in this text, just to avoid too much complexity.



Name	Date Modified
blue-empty.JPG	8/8/02
blue-katie-sitting.jpg	8/8/02
blue-katie-standing.jpg	8/8/02
blue-mark.jpg	8/8/02
c4.wav	7/2/02
concat.wav	7/5/02
croak.wav	7/10/02
Elliot-hello.wav	7/2/02

Format: All Files

Figure 2.4: File picker with media types identified

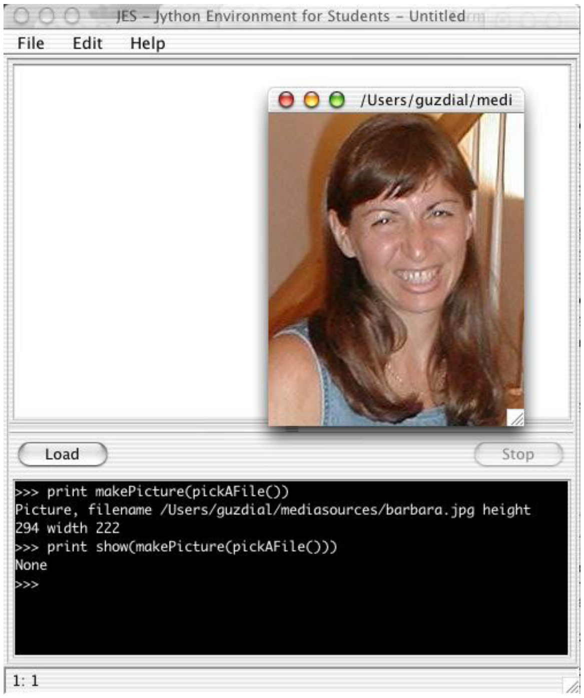
Showing a Picture

So now we know how to get a complete file name: Path and base name. This *doesn't* mean that we have the file itself loaded into memory. To get the file into memory, we have to tell JES how to interpret this file. *We* know that JPEG files are pictures, but we have to tell JES explicitly to read the file and make a picture from it. There is a function for that, too, named `makePicture`.

`makePicture` *does* require an *argument*—some input to the function. It takes a file name! Lucky us—we know how to get one of those!

```
>>> print makePicture(pickAFile())
Picture, filename /Users/guzdial/mediasources/barbara.jpg height
294 width 222
```

The result from `print` suggests that we did in fact make a picture, from a given filename and a given height and width. Success! Oh, you wanted to actually *see* the picture? We'll need another function! (Did I mention somewhere that computers are stupid?) The function to show the picture is named `show`. `show` *also* takes an argument—a `Picture`! Figure 2.5 is the result. Ta-dah!



```
>>> print makePicture(pickAFile())
Picture, filename /Users/guzdial/mediasources/barbara.jpg height
294 width 222
>>> print show(makePicture(pickAFile()))
None
>>>
```

Figure 2.5: Picking, making, and showing a picture

Notice that the output from `show` is `None`. Functions in Python don't *have* to return a value, unlike real mathematical functions. If a function *does* something (like opening up a picture in a window), then it doesn't also need to return a value. It's still pretty darn useful.

Playing a Sound

We can actually replicate this entire process with sounds.

- We still use `pickAFile` to find the file we want and get its file name.
- We now use `makeSound` to make a Sound. `makeSound`, as you might imagine, takes a file name as input.
- We will use `play` to play the sound. `play` takes a sound as input, but returns `None`.

Here are the same steps we saw previously with pictures:

```
>>> print pickAFile()
/Users/guzdial/mediasources/hello.wav
>>> print makeSound(pickAFile())
Sound of length 54757
>>> print play(makeSound(pickAFile()))
None
```

(We'll explain what the length of the sound means in the next chapter.) Please do try this on your own, using JPEG files and WAV files that you have on your own computer, that you make yourself, or that came on your CD. (We talk more about where to get the media and how to create it in future chapters.)

Congratulations! You've just worked your first media computation!

Naming your Media (and other Values)

`print play(makeSound(pickAFile()))` looks awfully complicated and long to type. You may be wondering if there are ways to simplify it. We can actually do it just the way that mathematicians have for centuries: We name the pieces! The results from functions can be named, and these names can be used in the inputs to other functions.

We name values by *assigning* names to values using an *equals* sign, `=`. In the below example, we assign names (called *variables* in this context) to each of the file name and to the picture.

```
>>> myfilename = pickAFile()
>>> print myfilename
/Users/guzdial/mediasources/barbara.jpg
>>> mypicture = makePicture(myfilename)
>>> print mypicture
Picture, filename /Users/guzdial/mediasources/barbara.jpg height
294 width 222
```

Notice that the algebraic notions of *subsitution* and *evaluation* work here as well. `mypicture = makePicture(myfilename)` causes the exact same picture to be created as if we had executed `makePicture(pickAFile())`[2], because we set `myfilename` to be equal to the result of `pickAFile()`. The values get substituted for the names when the expression is evaluated. `makePicture(myfilename)` is an expression which, at evaluation time, gets expanded into

[2] Assuming, of course, that you picked the same file.

makePicture("/Users/guzdial/mediasources/barbara.jpg") because "/Users/guzdial/mediasources/barbara.jpg" is the name of the file that was picked when pickAFile() was evaluated and the returned value was named myfilename.

We actually don't need to use print every time we ask the computer to do something. If we want to call a function that doesn't return anything (and so is pretty useless to print), we can just call the function by typing its name and its input (if any) and hitting return.

```
>>> show(mypicture)
```

We tend to call these statements to the computer that are telling it to do things *commands*. print mypicture is a command. So is myfilename = pickAFile(), and show(mypicture). These are more than expressions: They're telling the computer to *do* something.

2.2.3 Making a Recipe

We have now used names to stand for values. The values get substituted for the names when the expression is evaluated. We can do the same for recipes. We can name a series of commands, so that we can just use the name whenever we want the commands to be executed. This is exactly what defining a recipe or program is about.

Remember when we said earlier that just about anything can be named in computers? We've seen naming values. Now we'll see naming recipes.

Making it Work Tip: Try *every* recipe!
To really understand what's going on, type in, load, and execute *every* recipe in the book. *EVERY* one. None are long, but the practice will go along way to convincing you that the programs work and helping you understand why.

The name that Python understands as *defining* the name of new recipes is def. def isn't a function—it's a command like print. There are certain things that have to come after the word def, though. The structure of what goes on the line with the def command is referred to as the *syntax* of the command—the words and characters that have to be there for Python to understand what's going on, and the order of those things.

def needs three things to follow it on the same line:

- The name of the recipe that you're defining, like showMyPicture.

- Whatever *inputs* that this recipe will take. This recipe can be a function that takes inputs, like `abs` or `makePicture`. The inputs are named and placed between parentheses separated by commas. If your recipe takes no inputs, you simply enter `()` to indicate no inputs.

- The line ends with a colon, `:`.

What comes after that are the commands to be executed, one after the other, whenever this recipe is told to execute.

At this point, I need you to imagine a bit. Most real programs that do useful things, especially those that create user interfaces, require the definition of more than one function. Imagine that in the program area you have several `def` commands. How do you think Python will figure out that one function has ended and a new one begun? (Especially because it *is* possible to define functions *inside of* other functions.) Python needs some way of figuring out where the *function body* ends: Which statements are part of this function and which are part of the next.

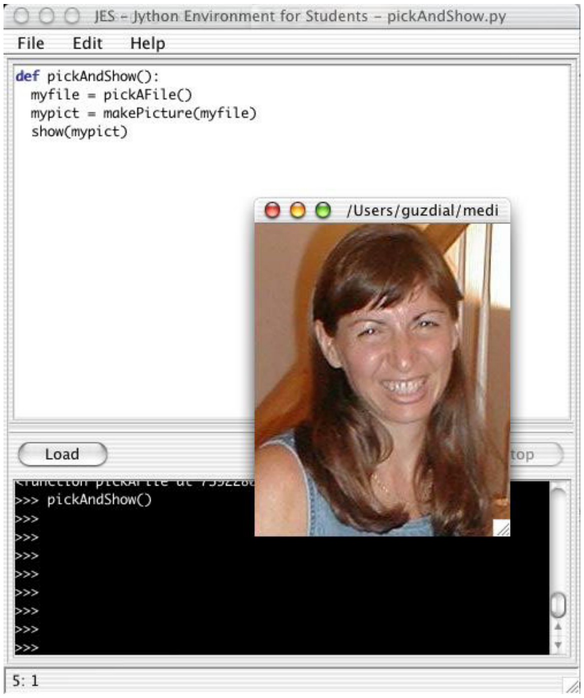
The answer is *indentation*. All the statements that are part of the definition are slightly indented after the `def` statement. I recommend using exactly two spaces—it's enough to see, it's easy to remember, and it's simple. You'd enter the function in the program area like this (where ␣ indicates a single space, a single press of the spacebar): `def hello():`
`␣␣print "Hello!"`

We can now define our first recipe! You type this into the program area of JES. When you're done, save the file: Use the extension ".py" to indicate a Python file. (I saved mine as `pickAndShow.py`.)

Recipe 2: Pick and show a picture

```
def pickAndShow():
  myfile = pickAFile()
  mypict = makePicture(myfile)
  show(mypict)
```

Once you've typed in your recipe and saved it, you can load it. Click the Load button.



```
def pickAndShow():
  myfile = pickAFile()
  mypict = makePicture(myfile)
  show(mypict)
```

```
<function pickAFile at 7392280
>>> pickAndShow()
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
```

Figure 2.6: Defining and executing `pickAndShow`

Debugging Tip: Don't forget to Load!
The most common mistake that I make with JES is typing in the function, saving it, then trying the function in the command area. You have to click the LOAD button to get it read in.

Now you can execute your recipe. Click in the command area. Since you aren't taking any input and not returning any value (i.e., this isn't a function), simply type the name of your recipe as a command:

```
>>> pickAndShow()
>>>
```

You'll be prompted for a file, and then the picture will appear (Figure 2.6).

We can similarly define our second recipe, to pick and play a sound.

Recipe 3: Pick and play a sound

```
def pickAndPlay():
  myfile = pickAFile()
  mysound = makeSound(myfile)
  play(mysound)
```

Making it Work Tip: Name the names you like

You'll notice that, in the last section, we were using the names `myfilename` and `mypicture`. In this recipe, I used `myfile` and `mypict`. Does it matter? Absolutely not! Well, to the computer, at any rate. The computer doesn't care what names you use—they're entirely for your benefit. Pick names that (a) are meaningful to you (so that you can read and understand your program), (b) are meaningful to others (so that others you show your program to can understand it), and (c) are easy to type. 25-character names, like, `myPictureThatIAmGoingToOpenAfterThis`) are meaningful, easy-to-read, but are a royal pain to type.

While cool, this probably isn't the most useful thing for you. Having to pick the file over-and-over again is just a pain. But now that we have the power of recipes, we can define new ones however we like! Let's define one that will just open a specific picture we want, and another that opens a specific sound we want.

Use `pickAFile` to get the file name of the sound or picture that you want. We're going to need that in defining the recipe to play that specific sound or show that specific picture. We'll just set the value of `myfile` *directly*, instead of as a result of `pickAFile`, by putting the string between quotes directly in the recipe.

Recipe 4: Show a specific picture

Be sure to replace `FILENAME` below with the complete path to your own picture file, e.g.,
”/Users/guzdial/mediasources/barbara.jpg”

```
def showPicture():
  myfile = "FILENAME"
  mypict = makePicture(myfile)
  show(mypict)
```

Recipe 5: Play a specific sound

Be sure to replace `FILENAME` below with the complete path to your own sound file, e.g.,
”/Users/guzdial/mediasources/hello.wav”.

```
def playSound():
  myfile = "FILENAME"
  mysound = makeSound(myfile)
  play(mysound)
```

Making it Work Tip: Copying and pasting
Text can be copied and pasted between the program and command areas. You can use `print pickAFile()` to print a filename, then select it and COPY it (from the EDIT menu), then click in the command area and PASTE it. Similarly, you can copy whole commands from the command area up to the program area: That's an easy way to test the individual commands, and then put them all in a recipe once you have the order right and they're working. You can also copy text within the command area. Instead of re-typing a command, select it, COPY it, PASTE it into the bottom line (make sure the cursor is at the end of the line!), and hit RETURN to execute it.

Variable Recipes: Real functions that Take Input

How do we create a real function with inputs out of our stored recipe? Why would you *want* to?

An important reason for using a variable so that input to the recipe can be specified is to make a program more *general*. Consider Recipe 4, `showPicture`. That's for a specific file name. Would it be useful to have a function that could take *any* file name, then make and show the picture? That kind of function handles the *general* case of making and showing pictures. We call that kind of generalization *abstraction*. Abstraction leads to general solutions that work in lots of situations.

Defining a recipe that takes input is very easy. It continues to be a matter of *substitution* and *evaluation*. We'll put a name inside those parentheses on the `def` line. That name is sometimes called the *parameter*—we'll just call it the *input variable* for right now.

When you evaluate the function, by specifying its name with an *input value* (also called the *argument*) inside parentheses (like `makepicture(myfilename)` or `show(mypicture)`), the input value is *assigned* to the input variable. During the execution of the function (recipe), the input value will be *substituted* for the value.

Here's what a recipe would look like that takes the file name as an input variable:

Recipe 6: Show the picture file whose file name is input

```
def showNamed(myfile):
  mypict = makePicture(myfile)
  show(mypict)
```

When I type
`showNamed("/Users/guzdial/mediasources/barbara.jpg")`
and hit return, the variable `myfile` takes on the value
`"/Users/guzdial/mediasources/barbara.jpg"`.
`myPict` will then be assigned to the picture resulting from reading and interpreting the file at
`‘‘/Users/guzdial/mediasources/barbara.jpg’’`
Then the pictures is shown.

We can do a sound in the same way.

Recipe 7: Play the sound file whose file name is input

```
def playNamed(myfile):
  mysound = makeSound(myfile)
  play(mysound)
```

We can also write recipes that take pictures or sounds in as the input values. Here's a recipe that shows a picture but takes the picture object as the input value, instead of the filename.

Recipe 8: Show the picture provided as input

```
def showPicture(mypict):
  show(mypict)
```

Now, what's the difference between the function `showPicture` and the provided function `show`? Nothing at all. We can certainly create a function that provides a new name to another function. If that makes your code easier for you to understand, than it's a great idea.

What's the *right* input value for a function? Is it better to input a filename or a picture? And what does "better" mean here, anyway? You'll read more about all of these later, but here's a short answer: Write the function that is most useful to you. If defining `showPicture` is more readable for you than `show`, then that's useful. If what you really want is a function that takes care of making the picture and showing it to you, then that's more useful and you might find the `showNamed` function the most useful.

Objects and Functions Summary

In this chapter, we talk about several kinds of encodings of data (or objects).

Integers (e.g., 3)	Numbers without a decimal point—they can't represent fractions.
Floating point numbers (e.g., 3.0, 3.01)	Numbers with a fractional piece to them.
Strings (e.g., "Hello!")	A sequence of characters (including spaces, punctuation, etc.) delimited on either end with a quote character.
File name	A filename is just a string whose characters represent a path, path delimiters, and a base file name.
Pictures	Pictures are encodings of images, typically coming from a JPEG file.
Sounds	Sounds are encodings of sounds, typically coming froma WAV file.

Here are the functions introduced in this chapter:

ord	Returns the equivalent numeric value (from the ASCII standard) for the input character.
abs	Takes input a number and returns the absolute value of it.
pickAFile	Lets the user pick a file and returns the complete path name as a string. No input.
makePicture	Takes a filename as input, reads the file, and creates a picture from it. Returns the picture.
show	Shows a picture provided as input. No return value.
makeSound	Takes a filename as input, reads the file, and creates a sound from it. Returns the sound.
play	Plays a sound provided as input. No return value.

Exercises

Exercise 9: Try some other operations with strings in JES. What happens if you multiple a number by a string, like `3 * "Hello"`? What happens if you try to multiply a string by a string, `"a" * "b"`?

Exercise 10: You can combine the sound playing and picture showing commands in the same recipe. Trying playing a sound and then show a picture while a sound is playing. Try playing a sound and opening several pictures while the sound is still playing.

Exercise 11: We evaluated the expression `pickAFile()` when we wanted to execute the function named `pickAFile`. But what is the name `pickAFile` anyway? What do you get if you `print pickAFile`? How about `print makePicture`? What do you think's going on here?

Exercise 12: Try the `playNamed` function. You weren't given any examples of its use, but you should be able to figure it out from `showNamed`.

To Dig Deeper

Part II

Sounds

Chapter 3

Encoding and Manipulating Sounds

3.1 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 these sounds into the numbers of a computer?

3.1.1 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 in the surface of the water (Figure 3.1). 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 are actually ones that arise from *combinations* of ripples—some waves are the sums and interactions from other waves.

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

Figure 3.1: Raindrops causing ripples in the surface of the water, just as sound causes ripples in the air

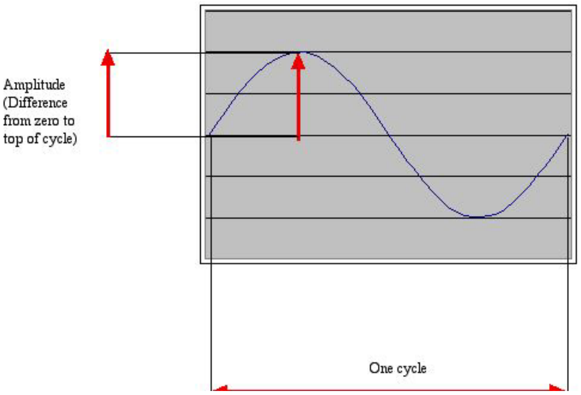


Figure 3.2: One cycle of the simplest sound, a sine wave

The simplest sound in the world is a *sine wave* (Figure 3.2). 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*.

Formally, amplitude is measured in Newtons per meter-squared (N/m^2). That's a rather hard unit to understand in terms of perception, but you can get a sense of the amazing range of human hearing from this unit. The smallest sound that humans typically hear is $0.0002N/m^2$, and the point at which we sense the vibrations in our entire body is $200N/m^2$! In general, amplitude is the most important factor in our perception of *volume*: If the amplitude rises, we perceive the sound as being louder typically. Other factors like air pressure factor into our perception of increased volume, too. Ever noticed how sounds sound differently on very humid days as compared with very dry days?

When we percieve an increase in volume, what we're really perceiving is the *intensity* of sound. Intensity is measured in watts per meter-squared (W/m^2). (Yes, those are watts just like the ones you're referring to when you get a 60-watt light bulb—it's a measure of power.) The intensity is proportional to the square of the amplitude. For example, if the amplitude doubles, intensity quadruples.

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 perception of sound is *logarithmically* related to the actual phenomena. Intensity is an example of this. A change in intensity from $0.1W/m^2$ to $0.01W/m^2$ sounds the *same* to us (as in the same amount of volume change) as a change in intensity of $0.001W/m^2$ to $0.0001W/m^2$.

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 does match the way we perceive volume. It's always a ratio, a comparison of two values. $10 * log_{10}(I_1/I_2)$ is change in intensity in decibels between 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 roubly 6dB.

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 intensity 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 cyle 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 (Figure 3.3). 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 5 Hz and 20,000 Hz (or 20 kiloHertz, abbreviated 20 kHz). Again, as with amplitudes, that's an enormous range! To give you

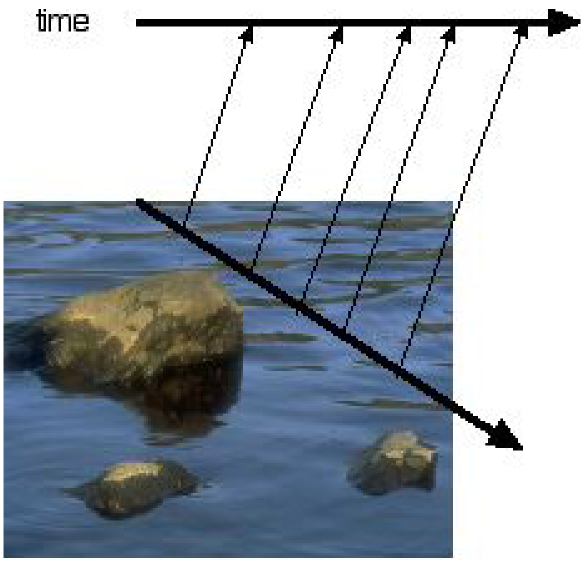


Figure 3.3: The distance between peaks in ripples in a pond are not constant—some gaps are longer, some shorter

a sense of where music fits into that spectrum, the note A above middle C is 440 Hz in traditional, *equal temperament* tuning (Figure 3.4).

Like intensity, our perception of pitch is almost exactly proportional to the log of the frequency. We really 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 percieve the same pitch change (or *pitch interval*) as a shift from 1000 Hz to 2000 Hz. Obviously, a different of 100 Hz is a lot smaller than a change of 1000 Hz, but we perceive it to be the same.

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 with respect to standards, but there are some universals. Among these universals are the use of pitch intervals (e.g., the ratio between notes C and D remains the same in every octave), the relationship between octaves remains constant, 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

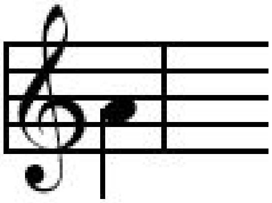


Figure 3.4: The note A above middle C is 440 Hz

is it that a flute playing a note sounds *so* different than 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 make different sounds (especially different instruments) different.

- Real sounds are almost never single frequency sound waves. Most natural sounds have *several* frequencies in them, often at different amplitudes. 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.

- Instrument sounds are not continuous with respect to amplitude and frequency. Some come slowly up to the target volume 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 represented very well 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 synthesizing with sine waves, but synthesizers sometimes also use other kinds of wave forms to get different kinds of sounds (Figure 3.5).

Exploring how sounds look

On your CD, you will find the *MediaTools* with documentation for how to get it started. The MediaTools contains tools for sound, graphics, and video.

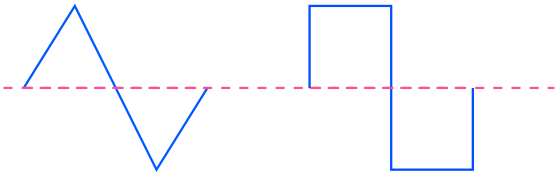


Figure 3.5: Some synthesizers using triangular (or *sawtooth*) or square waves.

Using the sound tools, you can actually observe sounds to get a sense of what louder and softer sounds look like, and what higher and lower pitched sounds look like.

The basic sound editor looks like Figure 3.6. 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!)

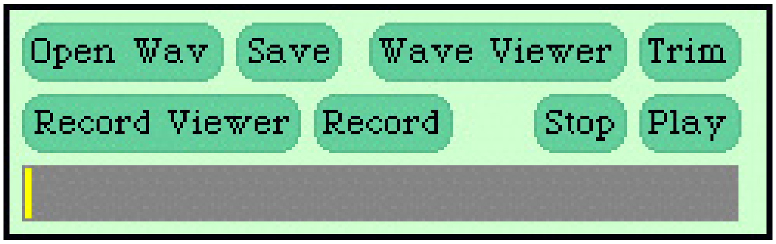


Figure 3.6: Sound editor main tool

To view sounds, click the VIEW 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 3.7). In the signal view, you're looking at the sound raw—each increase in air pressure results in an 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 the sounds' look changes. You can always get back to the signal view from another view by clicking the SIGNAL button.

The second view is the *spectrum view* (Figure 3.8). The spectrum view is a completely different perspective on the sound. In the previous section, you read that natural sounds are often actually composed of several different frequencies at once. The spectrum view shows these individual frequencies.

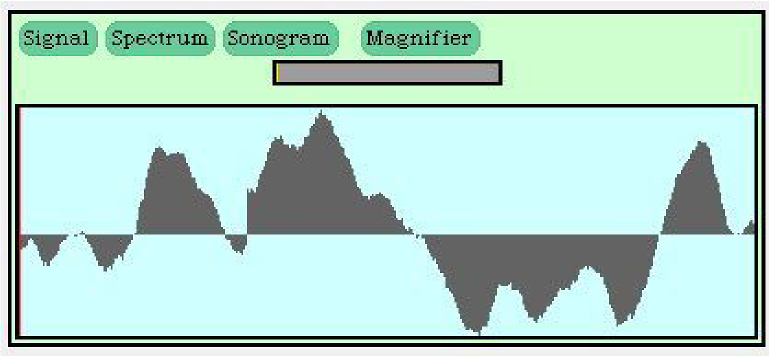


Figure 3.7: Viewing the sound signal as it comes in

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 3.9 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). The specific technique being used in the MediaTools signal view is a *Fast Fourier Transform* (or *FFT*), a very common way to do Fourier transforms quickly on a computer so that we can get a real time view of the changing spectra.

The third view is the *sonogram view* (Figure 3.10). The sonogram view is very much like the spectrum view in that it's describing 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). 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 in how different vocal sounds differ.

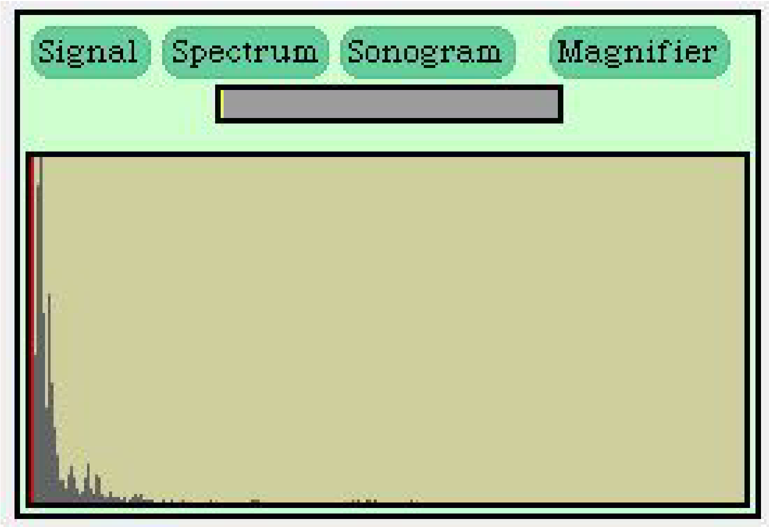


Figure 3.8: Viewing the sound in a spectrum view

Making it Work Tip: Explore sounds!
You really should try these different views on real sounds. You'll get a much better understanding for sound and what the manipulations we're doing in this chapter are doing to the sounds.

3.1.2 Encoding the Sound

You just read about how sounds work physically and how we perceive them. To manipulate these sounds on a computer and to play them back on a computer, we have to digitize them. To digitize sound means to take this flow of waves and turn it essentially 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 Wintel computers), then it's possible, at any moment, to measure the amount of air pressure against that microphone as a single number. Positive numbers correspond to rises in pressure, and negative

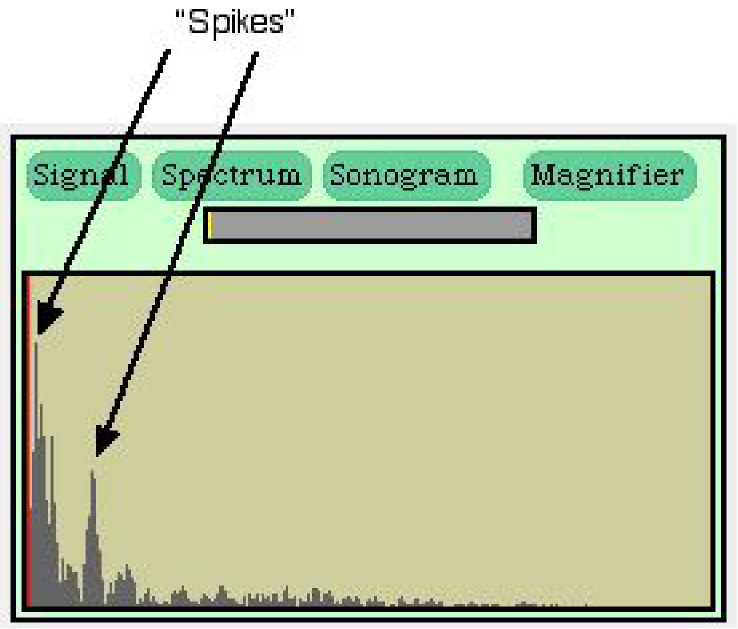


Figure 3.9: Viewing a sound in spectrum view with multiple "spikes"

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 continuous changing pressure wave. How do we store that in our computer?

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've had some calculus, you've got some idea of how we might do that. 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 3.11). With that idea, it's pretty clear that if we capture enough of those microphone pressure readings, we capture the wave. We call each of those pressure readings 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

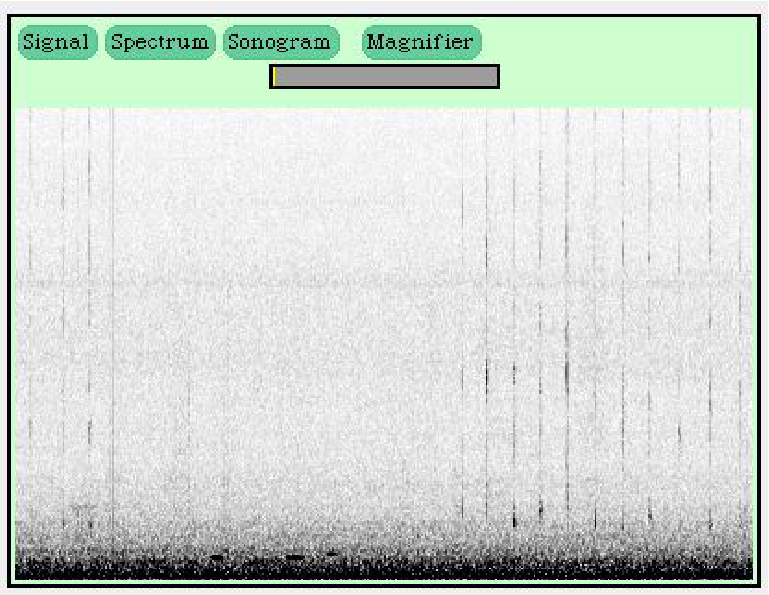


Figure 3.10: Viewing the sound signal in a sonogram view

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 8,000 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 8,000 cycles per second.

Computer Science Idea: Nyquist Theorem
To capture a sound of at most n cycles per second, you need to capture $2n$ 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 4,000 Hz. That's why our telephone system is designed around capturing 8,000 samples per second. That's why playing music through the telephone doesn't really work very well. The limits of (most) human hearing

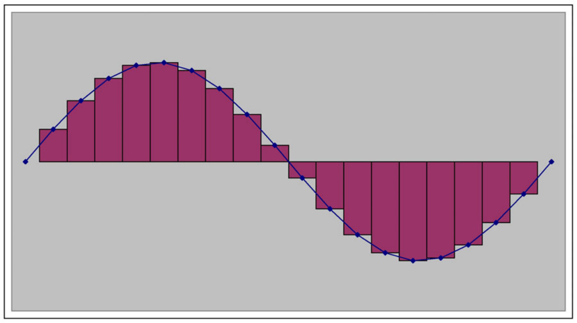


Figure 3.11: Area under a curve estimated with rectangles

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 it will sound quite reasonable. If you use too low of 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 two bytes, or 16 bits. Though there are larger *sample sizes*, 16 bits works perfectly well for most applications. CD-quality sound uses 16 bit samples.

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 those bits (remember, it's just 0 or 1) to represent whether we're talking about a positive (0) or negative (1) number. We call that 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
```

That looks forbidding. 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 takes up one of the 32,768 patterns.

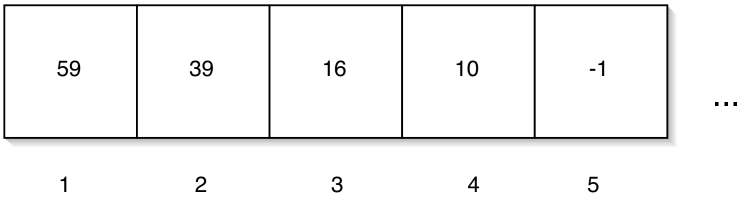
The sample size is a limitation on the amplitude of the sound that can be captured. If you have a sound that generates a pressure greater than 32,767 (or a rarefaction greater than -32,768), you'll only capture up to the limits of the 16 bits. If you were to look at the wave in the signal view, it would look like somebody took some scissors and *clipped* off the peaks of the waves. We call that effect *clipping* for that very reason. If you play (or generate) a sound that's clipped, it sound 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 is *pulse coded modulation (PCM)*. You may encounter that 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 into 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 one is 1, the second one is 2, and so on. You can think about an array as a long line of boxes, each one holding a value and each box having an index number on it (Figure 3.12).



59	39	16	10	-1	...
1	2	3	4	5	

Figure 3.12: A depiction of the first five elements in a real sound array

Using the MediaTools, you can graph a sound file (Figure 3.13) and get a sense of where the sound is quiet (small amplitudes), and loud (large amplitudes). This is actually important if you want to manipulate the sound. For example, the gaps between recordedwords tend to be quiet—at least quieter than the words themselves. You can pick out where words end by looking for these gaps, as in Figure 3.13.

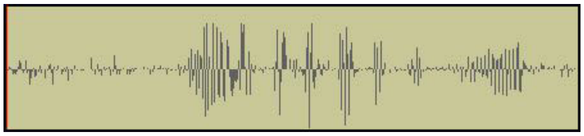


Figure 3.13: A sound recording graphed in the MediaTools

You will soon read about how to read a file containing a recording of a sound into a *sound object*, view the samples in that 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 elements in an array.

3.1.3 Using MediaTools for looking at captured sounds

The MediaTools for manipulating sounds that you read about earlier can also be used to study sound files. Any WAV file on your computer can be opened and studied within the sound tools.

From the basic sound editor tool, click on File to get the option to open a WAV file (Figure 3.14). The MediaTools' open file dialog will then appear. Find a WAV file by clicking on the directories on the left until you find one that contains the WAV files you want on the right (Figure 3.15), then click OK.

Your CD contains a `mediasources` directory on it. Most of the examples in the book use the media in this directory. You'll probably want to drag the `mediasources` d

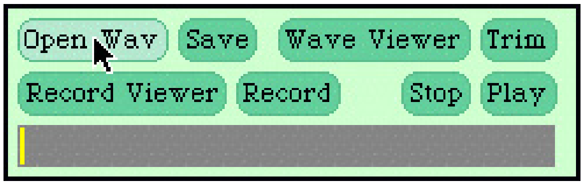
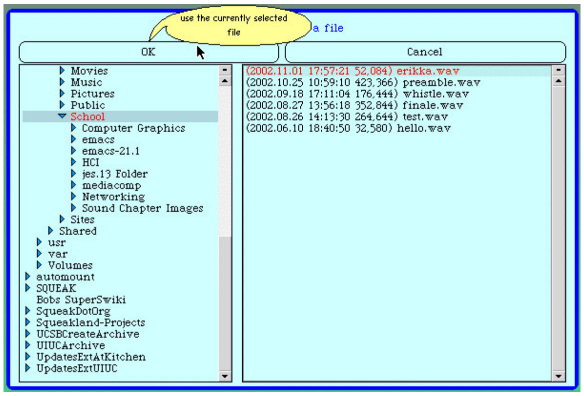


Figure 3.14: The sound editor open menu

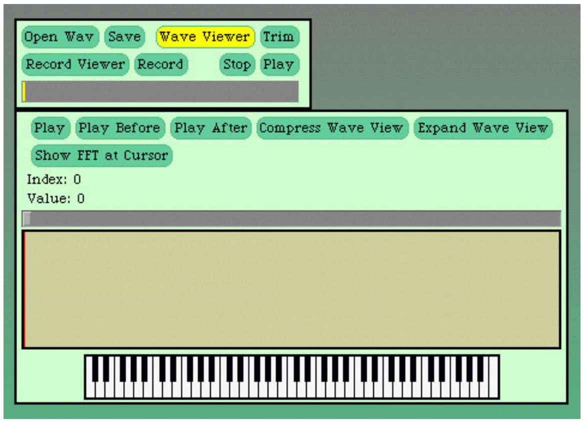


```
      Movies
      Music
      Pictures
      Public
      School
        Computer Graphics
        emacs
        emacs-21.1
        HCI
        jes.13 Folder
        mediacomp
        Networking
        Sound Chapter Images
      Sites
    Shared
  usr
  var
  Volumes
automount
SQUEAK
Bobs SuperSwiki
SqueakDotOrg
Squeakland-Projects
UCSBCreateArchive
UIUCArchive
UpdatesExtAtKitchen
UpdatesExtUIUC
```

```
(2002.11.01 17:57:21 52,084) erikka.wav
(2002.10.25 10:59:10 423,366) preamble.wav
(2002.09.18 17:11:04 176,444) whistle.wav
(2002.08.27 13:56:18 352,844) finale.wav
(2002.08.26 14:13:30 264,644) test.wav
(2002.06.10 18:40:50 32,580) hello.wav
```

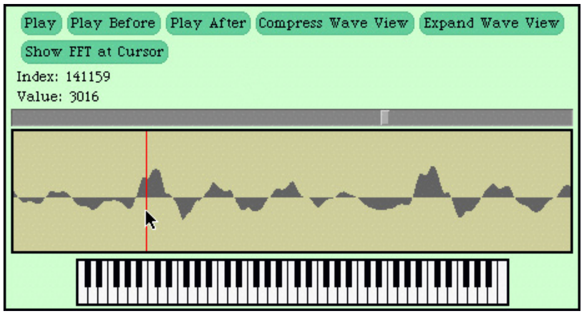
Figure 3.15: MediaTools open file dialog

You will then be shown the file in the sound editor view (Figure 3.16). The sound editor lets you explore a sound in many ways (Figure 3.17). As you scroll through the sound and change the *sound cursor* (the red/blue line in the graph) position, the INDEX changes to show you which sound array element you're currently looking at, and the VALUE shows you the value at that index. You can also fit the whole sound into the graph to get an overall view (but currently breaks the index/value displays). You can even "play" your recorded sound as if it were an instrument—try pressing the piano keys across the bottom of the editor. You can also set the cursor (via the scrollbar or by dragging in the graph window) and play the sound before the cursor—a good way to hear what part of the sound corresponds to what index positions. Clicking the <> button provides a menu of option which includes getting an FFT view of the sound.



Index: 0
Value: 0

Figure 3.16: A sound opened in the editor



Index: 141159
Value: 3016

Figure 3.17: Exploring the sound in the editor

3.2 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 a filename of a WAV file, and make a sound from it. You already saw how to do that in a previous chapter.

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.

4. You may then want to write the sound back out to a new file, to use elsewhere. (Most sound editing programs know how to deal with audio.)

3.2.1 Open sounds and manipulating samples

You have already seen how to find a file with `pickAFile` and then making a sound object with `makeSound`. Here's an example of doing that in JES.

```
>>> filename=pickAFile()
>>> print filename
/Users/guzdial/mediasources/preamble.wav
>>> sound=makeSound(filename)
>>> print sound
Sound of length 421109
```

You can get the samples from a sound using `getSamples`. The function `getSamples` takes a sound as input and returns an array of all the samples as sample objects. When you execute this function, it may take quite a while before it finishes—longer for longer sounds, shorter for shorter sounds.

```
>>> samples=getSamples(sound)
>>> print samples
Samples, length 421109
```

The function getSamples is making an array of sample *objects* out of the basic sample array. An *object* is more than just a simple value like you read about earlier—for one difference, a sample object also knows what sound it came from and what its index is. You will read more about objects later, but take it at face value now that getSamples provides you with a bunch of sample objects that you can manipulate—and in fact, makes manipulation pretty easy. You can get the value of a sample object by using getSample (with a sample object as input), and you set the sample value with setSample (with a sample object and a new value as input).

But before we get to the manipulations, let's look at some other ways to get and set samples. We can ask the sound to give us the values of specific samples at specific indices, by using the function getSampleValueAt. The input values to getSampleValueAt are a sound and an index number.

```
>>> print getSampleValueAt(sound,1)
36
>>> print getSampleValueAt(sound,2)
29
```

What numbers can we use as index values? Anything between 1 and the length of the sound in samples. We get that length (the maximum index value) with getLength. Notice the error that we get below if we try to get a sample past the end of the array.

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

We can similarly change sample values by using setSampleValueAt. It also takes as input values a sound, and an index, but also a new value for the sample at that index number. We can then check it again with getSampleValueAt.

```
>>> print getSampleValueAt(sound,1)
36
>>> setSampleValueAt(sound,1,12)
>>> print getSampleValueAt(sound,1)
12
```

Common Bug: Mistyping a name
You just saw a whole bunch of function names, and some of them are pretty long. What happens if you type one of them wrong? JES will complain that it doesn't know what you mean, like this:

```
>>> writeSndTo(sound,"mysound.wav")
A local or global name could not be
found.
```

It's no big deal. JES will let you copy the mistyped command, paste it into the bottommost line of the command area, then fix it. Be sure to put the cursor at the end of the line before you press the Return key.

What do you think would happen if we then played this sound? Would it really sound different than it did before, now that we've turned the first sample from the number 36 to the number 12? Not really. To explain why not, let's find out what the sampling rate is for this sound, by using the function `getSamplingRate` which takes a sound as its input.

```
>>> print getSamplingRate(sound)
22050.0
```

The sound that we're manipulating in this examples (a recording of me reading part of the U.S. Constitution's preamble) has a sampling rate of 22,050 samples per second. Changing one sample changes 1/22050 of the first second of that sound. If you can hear that, you have amazingly good hearing—and I will have my doubts about your truthfulness!

Obviously, to make a significant manipulation to the sound, we have to manipulate hundreds if not thousands of samples. We're certainly not going to do that by typing thousands of lines of

```
setSampleValueAt(sound,1,12)
setSampleValueAt(sound,2,24)
setSampleValueAt(sound,3,100)
setSampleValueAt(sound,4,99)
setSampleValueAt(sound,5,-1)
```

We need to take advantage of the computer executing our recipe, by telling it to go do something hundreds or thousands of times. That's the topic for the next section.

But we will end this section by talking about how to write your results back out to a file. Once you've manipulated your sound and want to save it out to use elsewhere, you use `writeSoundTo` which takes a sound and a new filename as input. Be sure that your file ends with the extension ".wav" if you're saving a sound so that your operating system knows what to do with it!

```
>>> print filename
/Users/guzdial/mediasources/preamble.wav
>>> writeSoundTo(sound,"/Users/guzdial/mediasources/new-preamble.wav")
```

Common Bug: Saving a file quickly—and losing it!
What if you don't know the whole path to a directory of your choosing? You don't have to specify anything more than the base name.

```
>>> writeSoundTo(sound,"new-preamble.wav")
```

The problem is finding the file again! In what directory did it get saved? This is a pretty simple bug to resolve. The default directory (the one you get if you don't specify a path) is wherever JES is. Find JES and you'll find `new-preamble.wav` (in this example).

You'll probably figure out when playing sounds a lot that if you use `play` a couple times in quick succession, you'll mix the sounds. How do you make sure that the computer plays only a single sound and then waits for that sound to end? You use something called `blockingPlay`. That works the same as `play`, but it waits for the sound to end so that no other sound can interfere while it's playing.

3.2.2 Introducing the loop

The problem we're facing is a common one in computing: How do we get the computer to do something over-and-over again? We need to get the computer to *loop* or *iterate*. Python has commands especially for looping (or iterating). We're mostly going to use the command `for`. A `for` loop executes some commands (that you specify) for an array (that you provide), where each time that the commands are executed, a particular variable (that you, again, get to name) will have the value of a different element of the array.

We are going to use the `getSamples` function we saw earlier to provide our array. We will use a `for` loop that looks like this:

```
for sample in getSamples(sound):
```

Let's talk through the pieces here.

- First comes the command name `for`.
- Next comes the variable name that you want to use in your code for addressing (and manipulating) the elements of the array.
- The word `in` is **required**—you must type that! It makes the command more readable than leaving it out, so there's a benefit to the extra four keystrokes (space-i-n-space).
- Then, you need the array. We use the function `getSamples` to generate an array for us.
- Finally, you need a colon (":"). The colon is important—it signifies that what comes next is a *block* (you should recall reading about them in Section 2.2.3 on 32).

What comes next are the commands that you want to execute for each sample. Each time the commands are executed, the variable (in our example `sample`) will be a different element from the array. The commands (called the *body*) are specified as a block. This means that they should follow the `for` statement, each on their own line, *and indented by two more spaces!* For example, here is the `for` loop that simply sets each sample to its own value (a particularly useless exercise, but it'll get more interesting in just a couple pages).

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

Let's talk through this code.

- The first statement says that we're going to have a `for` loop that will set the variable `sample` to each of the elements of the array that is output from `getSamples(sound)`.
- The next statement is indented, so it's part of the body of the `for` loop—one of the statements that will get executed each time `sample` has a new value. It says to name the value of the sample in the variable `sample`. That name is `value`.
- The third statement is still indented, so it's still part of the loop body. Here we set the value of the sample to the value of the variable `value`.

Here's the exact same code (it would work *exactly* the same), but with different variable names.

```
for s in getSamples(sound):
  v = getSample(s)
  setSample(s,v)
```

What's the difference? Slightly easier to confuse variable names. `s` and `v` are not as obvious what they are naming as `sample` and `value`. Python doesn't care which we use, and the single character variable names are clearly easier to type. But the longer variable names make it easier to understand your code later.

Notice that the earlier paragraph had emphasized by two *more* spaces. Remember that what comes after a function definition `def` statement is *also* a block. If you have a `for` loop inside a function, then the `for` statement is indented two spaces already, so the body of the `for` loop (the statements to be executed) must be indented *four* spaces. The `for` loop's block is inside the function's block. That's called a *nested block*—one block is nested inside the other. Here's an example of turning our useless loop into an equally useless function:

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

You don't actually have to put loops into functions to use them. You can type them into the command area of JES. JES is smart enough to figure out that you need to type more than one command if you're specifying a loop, so it changes the prompt from `>>>` to `...`. Of course, it can't figure out when you're done, so you'll have to just hit return without typing anything else to tell JES that you're done with the body of your loop. It looks something like this:

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

You probably realize that we don't really need the variable `value` (or `v`). We can combine the two statements in the body into one. Here's doing it at the command line:

```
>>> for sample in getSamples(sound):
...   setSample(sample,getSample(sample))
```

Now that we see how to get the computer to do thousands of commands without writing thousands of individual lines, let's do something useful with this.

Common Bug: Keep sounds short
Don't work with sounds over 10 seconds. You could have problems with memory, but much worse, there's a bug in JES where the sound can get hideously noisy past 10 seconds.

Common Bug: Windows and WAV files
The world of WAV files isn't as compatible and smooth as one might like. WAV files saved from JES play fine in JES or any Java-based application. MediaTools handles them fine, as does Apple QuickTime, which is available for free for both Windows and Macintosh computers[1]. But WinAmp 2 (WinAmp 3 seems to be better) and Windows Media Player cannot play WAV files generated from JES. It seems that Sun's definition of WAV files in Java, and Microsoft's definition of WAV files differ. Imagine that....
We don't yet have a complete and free solution to this problem. If you open the WAV file in the MediaTools, then save it back out to a WAV file (an option under the File menu), you can definitely open the file in WinAmp 3. A complete solution is to purchase QuickTime Player Pro (for about $50). If you open a JES-saved WAV file from QuickTime Player Pro, then Export the file back out as a WAV file, you will get a WAV file that works correctly in WinAmp and Windows Media Player.

3.2.3 Changing the volume of sounds

Earlier, we said that the amplitude of a sound is the main factor in the volume. This means that if we increase the amplitude, we increase the volume. Or if we decrease the amplitude, we decrease the volume.

Don't get confused here — changing the amplitude doesn't reach out and twist up the volume knob on your speakers. If your speaker's volume (or computer's volume) is turned down, the sound will never get very loud. The point is getting the sound itself louder. Have you ever watched a movie on TV where, without changing the volume on the TV, sound becomes so low that you can hardly hear it? (Marlon Brando's dialogue in the movie *The Godfather* comes to mind.) That's what we're doing here. We can make sounds *shout* or *whisper* by tweaking the amplitude.

Increasing volume

Here's a function that doubles the amplitude of an input sound.

Recipe 9: Increase an input sound's volume by doubling the amplitude

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

Go ahead and type the above into your JES Program Area. Click Load to get Python to process the function and make the name `increaseVolume` stand for this function. Follow along the example below to get a better idea of how this all works.

To use this recipe, you have to create a sound first, then pass it in as input. In the below example, we get the filename by setting the variable `f` explicitly to a string that is a filename, as opposed to using `pickAFile`. Don't forget that you can't type this code in and have it work as-is: Your path names will be different than mine!

```
>>> f="/Users/guzdial/mediasources/gettysburg10.wav"
>>> s=makeSound(f)
>>> increaseVolume(s)
```

We then create a sound that we name `s`. When we evaluate `increaseVolumes`, the sound that is named `s` becomes *also* named `sound`, within the function `increaseVolume`. This is a *very* important point. **Both names refer to the same sound!** The changes that take place in `increaseVolume` are really changing the *same* sound. You can think of each name as an *alias* for the other: They refer to the same basic *thing*.

There's a side point to mention just in passing, but becomes more important later: When the function `increaseVolume` ends, the name `sound` *has no value*. It only exists during the duration of that function's execution. We say that it only exists within the *scope* of the function `increaseVolume`.

We can now play the file to hear that it's louder, and write it to a new file.

```
>>> play(s)
>>> writeSoundTo(s,"/Users/guzdial/mediasources/louder-g10.wav")
```

Did that really work?

Now, is it really louder, or does it just seem that way? We can check it in several ways. You could always make the sound even louder by evaluating `increaseVolume` on our sound a few more times—eventually, you'll be totally convinced that the sound is louder. But there are ways to test even more subtle effects.

If we compared graphs of the two sounds, you'd find that the sound in the new file (`louder-g10.wav` in our example) has much bigger amplitude than the sound in the original file (`gettysburg10.wav`, which is in the `mediasources` directory on your CD). Check it out in Figure 3.18.

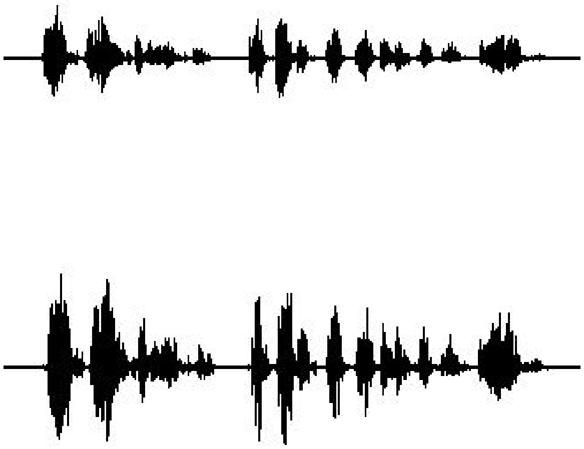
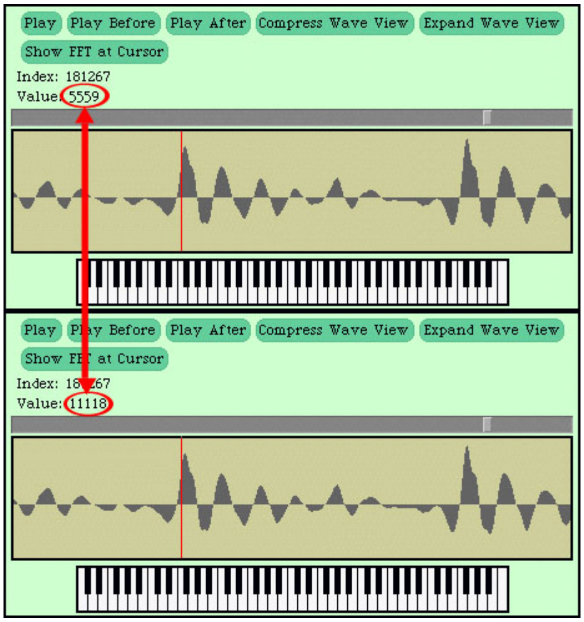


Figure 3.18: Comparing the graphs of the original sound (top) and the louder one (bottom)

Maybe you're unsure that you're really seeing a larger wave in the second picture. (It's particularly hard to see in the MediaTools which automatically scales the graphs so that they look the same—I had to use an another sound tool to generate the pictures in Figure 3.18.) You can use the MediaTools to check the individual sample values. Open up both WAV files, and open the sound editor for each. Scroll down into the middle of the sound, then drag the cursor to any value you want. Now, so the same to the second one.



Index: 181267
Value: 5559

Index: 181267
Value: 11118

Figure 3.19: Comparing specific samples in the original sound (top) and the louder one (bottom)

You'll see that the louder sound (bottom one in Figure 3.19) really does have double the value of the same sample in the original sound.

Finally, you can always check for yourself from within JES. If you've been following along with the example[2], then the variable `s` is the now louder sound. `f` should still be the filename of the original sound. Go ahead and make a new sound object which is the *original* sound—that is named below as `soriginal` (for *sound original*). Check any sample value that you want—it's always true that the louder sound has twice the sample values of the original sound.

```
>>> print s
Sound of length 220567
>>> print f
/Users/guzdial/mediasources/gettysburg10.wav
>>> soriginal=makeSound(f)
>>> print getSampleValueAt(s,1)
```

[2] What? You haven't? You *should*! It'll make much more sense if you try it yourself!

```
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
```

That last one is particularly telling. Even negative values become *more* negative. That's what's meant by "increasing the amplitude." The amplitude of the wave goes in *both* directions. We have to make the wave larger in both the positive and negative dimensons.

It's important to do what you just read in this chapter: *Doubt* your programs. Did that *really* do what I wanted it to do? The way you check is by *testing*. That's what this section is about. You just saw several ways to test:

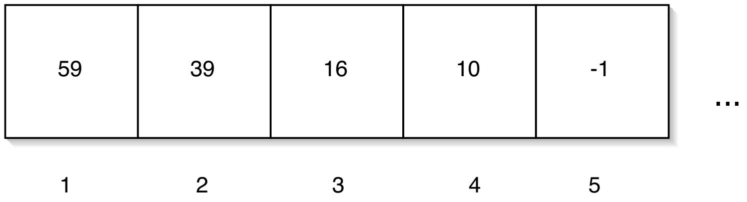
- By looking at the result overall (like with the graphs),
- By checking at pieces of the results (like with the MediaTools), and
- By writing additional code statements that check the results of the original program.

How did it work?

Let's walk through the code, slowly, and consider how this program worked.

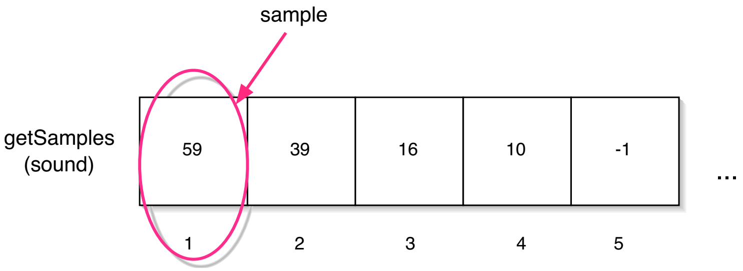
```
def increaseVolume(sound):
  for sample in getSamples(sound):
    value = getSample(sample)
    setSample(sample,value * 2)
```

Recall our picture of how the samples in a sound array might look.

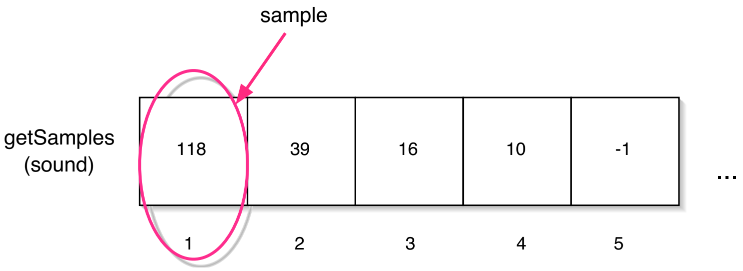


This is what `getSamples(sound)` would return: An array of sample values, each numbered. The `for` loop allows us to walk through each sample, one at a time. The name `sample` will be assigned to each sample in turn.

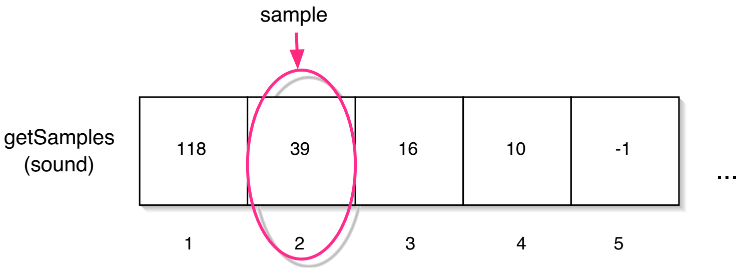
When the `for` loop begins, `sample` will be the name for the first sample.



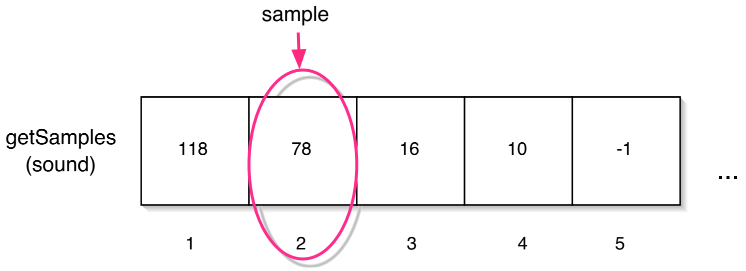
The variable `value` will take on the value of 59 when `value=getSample(sample)` is executed. The sample that the name `sample` references will then be doubled with `setSample(sample,value*2)`.



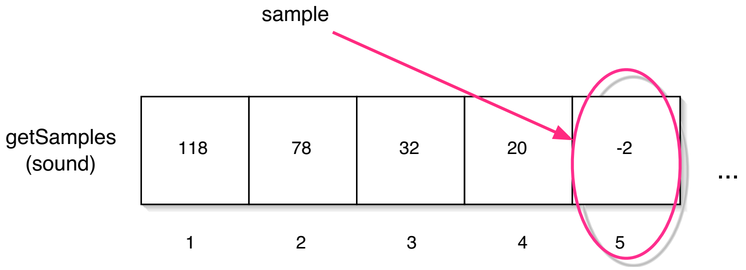
That's the end of the first pass through the body of the `for` loop. Python will then start the loop over and move `sample` on to point at the *next* element in the array.



Again, the `value` is set to the value of the sample, then the sample will be doubled.



Again, the loop repeats through the five samples pictured.



But really, the `for` loop *keeps* going through all the samples—tens of thousands of them! Thank goodness it's the *computer* executing this recipe!

One way to think about what's happening here is that the `for` loop doesn't really *do* anything, in the sense of changing anything in the sound. Only the *body* of the loop does work. The `for` loop tells the computer *what* to do. It's a manager. What the computer actually does is something like this:

```
sample = sample #1
value = value of the sample, 59
change sample to 118
sample = sample #2
value = 39
change sample to 78
sample = sample #3
...
sample = sample #5
value = -1
change sample to -2
...
```

The `for` loop is only saying, "Do all of this for every element in the array." It's the *body* of the loop that contains the Python commands that get executed.

What you have just read in this section is called *tracing* the program. We slowly went through how each step in the program was executed. We drew pictures to describe the data in the program. We used numbers, arrows, equations, and even plain English to explain what was going on in the program. This is the single most important technique in programming. It's part of *debugging*. Your program will *not* always work. Absolutely, guaranteed, without a shadow of a doubt—you will write code that does not do what you want. But the computer *will* do *SOMETHING*. How do you figure out what it *is* doing? You debug, and the most significant way to do that is by tracing the program.

Decreasing volume

Decreasing volume, then, is the reverse of the previous process.

Recipe 10: Decrease an input sound's volume by halving the amplitude

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

We can use it like this.

```
>>> f=pickAFile()
>>> print f
/Users/guzdial/mediasources/louder-g10.wav
>>> sound=makeSound(f)
>>> print sound
Sound of length 220568
>>> play(sound)
>>> decreaseVolume(sound)
>>> play(sound)
```

We can even do it again, and lower the volume even further.

```
>>> decreaseVolume(sound)
>>> play(sound)
```

Normalizing sounds

If you think about it some, it seems strange that the last two recipes work! We can just multiply these numbers representing a sound—and the sound seems (essentially) the same to our ears? The way we experience a sound depends less on the specific numbers than on the *relationship* between them. Remember that the overall shape of the sound waveform is dependent on *many* samples. In general, if we multiply all the samples by the same multiplier, we only effect our sense of volume (intensity), not the sound itself. (We'll work to change the sound itself in future sections.)

A common operation that people want to do with sounds is to make them as **LOUD AS POSSIBLE**. That's called *normalizing*. It's not really hard to do, but it takes more lines of Python code than we've used previously and a few more variables, but we can do it. Here's the recipe, in English, that we need to tell the computer to do.

- We have to figure out what the largest sample in the sound is. If it's already at the maximum value (32767), then we can't really increase the volume and still get what seems like the same sound. Remember that we have to multiply all the samples by the same multiplier.

  It's an easy recipe (*algorithm*) to find the largest value—sort of a *sub-recipe* within the overall normalizing recipe. Define a name (say, `largest`) and assign it a small value (0 works). Now, check all the samples. If you find a sample larger than the `largest`, make that larger value the new meaning for `largest`. Keep checking the samples, now comparing to the *new* largest. Eventually, the very largest value in the array will be in the variable `largest`.

  To do this, we'll need a way of figuring out the maximum value of two values. Python provides a built-in function called `max` that can do that.

  ```
  >>> print max(8,9)
  9
  >>> print max(3,4,5)
  5
  ```

- Next, we need to figure out what value to multiply all the samples by. We want the largest value to become 32767. Thus, we want to figure out a *multiplier* such that

  $(multiplier)(largest) = 32767.$

  Solve for the multiplier:

  $multiplier = 32767/largest$. The multiplier will need to be a floating point number (have a decimal component), so we need to convince Python that not everything here is an integer. Turns out that that's easy—use 32767.0.

- Now, loop through all the samples, as we did for `increaseVolume`, and multiply the sample by the multiplier.

Here's a recipe to normalize sounds.

Recipe 11: Normalize the sound to a maximum amplitude

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

    print "Largest sample value in original sound was",
largest
    print "Multiplier is", multiplier

    for s in getSamples(sound):
        louder =  multiplier * getSample(s)
        setSample(s,louder)
```

There are several notational items to note about this program.

- There are blank lines in there! Sure, Python doesn't care about those. Adding blank lines can be useful to break up and improve the understandability of longer programs.

- There are `print` statements in there! Sure, `print` statements can be *really* useful. First, they give you some feedback that the program is running—a useful thing in long-running programs. Second, they show you what it's finding, which can be interesting. Third, it's a terrific testing method and a way to debug your programs. Let's imagine that the printout showed that the multiplier was less than 1.0. We know that that kind of multiplier *decreases* volume. You should probably suspect that something went wrong.

- Some of the statements in this recipe are pretty long, so they wrap around in the text. *Type them as a single line!* Python doesn't let you hit return until the end of the statement—make sure that your print statements are all on one line.

Here's what the program looks like running.

```
>>> normalize(sound)
Largest sample  in original sound was 5656
Multiplier is 5.7933168316831685
>>> play(sound)
```

Exciting, huh? Obviously, the interesting part is hearing the much louder volume, which is awfully hard to do in a book.

3.2.4 Manipulating different sections of the sound differently

These are useful things to do to sounds overall, but really interesting effects come from chopping up sounds and manipulating them differentially: Some words this way, other sounds that way. How would you do that? We need to be able to loop through *portions* of the sample, without walking through the whole thing. Turns out to be an easy thing to do, but we need to manipulate samples somewhat differently (e.g., we have to use index numbers) and we have to use our `for` loop in a slightly different way.

Recall that each sample has a number, and that we can get each individual sample with `getSampleValueAt` (with a sound and an index number as input). We can set any sample with `setSampleValueAt` (with inputs of a sound, an index number, and a new value). That's how we can manipulate samples without using `getSamples` and sample objects. But we still don't want to have to write code like:

```
setSampleAt(sound,1,12)
setSampleAt(sound,2,28)
...
```

Not for tens of thousands of samples!

What we need is to get the computer to address each of those samples, in turn, by index number. We need to get the `for` loop to go from 1 to 20,000-something (or whatever the length of the sound may be). As you might expect, Python does have a way of doing this. It's called the function `range`. `range` takes two inputs and returns an array of the integers between the two numbers—including the first one, but stopping before the last one. Some examples will help to make clearer what it does.

```
>>> 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, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,
83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99]
```

You might be wondering what this square bracket stuff is, e.g., `[1,2]` in the first example above. That's the notation for an array—it's how Python prints out a series of numbers to show that this is an array[3]. If we use `range` to generate the array for the `for` loop, our variable will walk through each of the sequential numbers we generate.

It turns out that `range` can also take *three* inputs. If a third input is provided, it's an *increment*—the amount to step between generated integers.

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

[3] Technically, `range` returns a *sequence*, which is a somewhat different ordered collection of data from an array. But for our purposes, we'll call it an array.

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45,
48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93,
96, 99]
```

Using `range`, we can do everything that we were doing with `getSamples`, but now directly referencing the index numbers. Here's Recipe 9 (page 66) written using `range`.

Recipe 12: Increase an input sound's volume by doubling the amplitude, using `range`

```
def increaseVolumeByRange(sound):
  for sampleIndex in range(1,getLength(sound)+1):
    value = getSampleValueAt(sound,sampleIndex)
    setSampleValueAt(sound,sampleIndex,value * 2)
```

Try it—you'll find that it performs just like the previous one.

But now we can do some really odd things with sounds, because we can control which samples that we're talking to. The below recipe *increases* the sound for the first half of the sound, then *decreases* it in the second half. See if you can trace how it's working.

Recipe 13: Increase the volume in the first half of the sound, and decrease in the second half

```
def increaseAndDecrease(sound):
  for sampleIndex in range(1,getLength(sound)/2):
    value = getSampleValueAt(sound,sampleIndex)
    setSampleValueAt(sound,sampleIndex,value * 2)
  for sampleIndex in range(getLength(sound)/2,getLength(sound)+1):
    value = getSampleValueAt(sound,sampleIndex)
    setSampleValueAt(sound,sampleIndex,value * 0.2)
```

Another way of writing array references

It's worth pointing out that, in many languages, square brackets ([ ]) are central to manipulating arrays. It's a standard notation in mnay languages for manipulating arrays. In fact, it works the same here in Python. For any array, `array[index]` returns the index-th element in the array. The number inside the square brackets is always an index variable, but it's sometimes referred to as a *subscript*, because of the way that mathematicians refer to the i-th element of a, e.g., a_i.

There is one catch from what we've been doing earlier: Arrays in Python are traditional computer science arrays. The first index is zero. The media functions built-in to JES allow you to think in terms of starting with 1, like most normal human endeavors. But when you do the square brackets, you're dealing with raw Python. Some of your results may feel like they're off-by-one.

Let's do it here with samples to demonstrate.

```
>>> samples = getSamples(sound)
>>> print samples[1]
Sample at 2 value at 696
>>> print samples[0]
Sample at 1 value at 0
>>> print samples[22000]
Sample at 22001 value at 22364
```

To demonstrate it in ways that you can trust the result (because you don't really know what's in the sound in the above examples), let's use `range` to make an array, then reference it the same way.

```
>>> myArray = range(0,100)
>>> print myArray[1]
1
>>> print myArray[0]
0
>>> print myArray[35]
35
>>> mySecondArray = range(0,100,2)
>>> print mySecondArray[35]
70
```

3.2.5 Splicing sounds

Splicing sounds is a term that dates back to when sounds were recorded on tape, so juggling the order of things on the tape involved literally cutting the tape into segments and then gluing it back together in the right order. That's "splicing". When everything is digital, it's *much* easier.

To splice sounds, we simply have to copy elements around in the array. It's easiest to do this with two (or more) arrays, rather than copying within the same array. If you copy all the samples that represent someone saying the word "The" up to the beginning of a sound (starting at index number 1), then you make the sound start with the word "The." Splicing lets you create all kinds of sounds, speeches, non-sense, and art.

The first thing to do in splicing is to figure out the index numbers that delimit the pieces you're interested in. Using the MediaTools, that's pretty easy to do.

- Open your WAV file in the MediaTools sound tools.
- Open the editor.
- Scroll and move the cursor (by dragging in the graph) until you think that the cursor is before or after a sound of interest.
- Check your positioning by playing the sound before and after the cursor, using the buttons in the sound editor.

Using exactly this process, I found the ending points of the first few words in `preamble10.wav`. (I figure that the first word starts at the index 1, though that might not always be true for every sound.)

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

Writing a loop that copies things from one array to another requires a little bit of juggling. You need to think about keeping track of two indices: Where you are in the array that you're copying *from*, and where you are in the array that you're copying *to*. These are two different variables, tracking two different indexes. But they both increment in the same way.

The way that we're going to do it (another *sub-recipe*) is to use one index variable to point at the right entry in the *target* array (the one that we're copying *to*), use a `for` loop to have the second index variable move across the right entries in the *source* array (the one that we're copying *from*), and (*very important!*) move the target index variable each time that we do a copy. This is what keeps the two index variables synchronized (or "in synch" but not "N*sync").

The way that we make the target index move is to add one to it. Very simply, we'll tell Python to do `targetIndex = targetIndex + 1`. If you're a mathematician, that probably looks non-sensical. "How can any variable equal itself plus one?" It's never true that $x = x+1$. But remember that "`=`" doesn't assert that the two sides are equal—it means "Make the name on the left stand for the value on the right." Thus, `targetIndex = targetIndex + 1` makes a lot of sense: Make the name `targetIndex` now be whatever `targetIndex` currently is plus one. That moves the target index. If we put this in the body of the loop where we're changing the source index, we'll get them moving in synchrony.

The general form of the sub-recipe is:

```
targetIndex = Where-the-incoming-sound-should-start

for sourceIndex in range(startingPoint,endingPoint)

  setSampleValueAt( target, targetIndex, getSampleValueAt( source,
sourceIndex))

  targetIndex = targetIndex + 1
```

xxx Would some example pictures help here?

Below is the recipe that changes the preamble from "We the people of the United States" to "We the UNITED people of the United States."

Recipe 14: Splice the preamble to have united people

Be sure to change the `file` variable before trying this on your computer.

```
# Splicing
# Using the preamble sound, make "We the united people"
def splicePreamble():
file = "/Users/guzdial/mediasources/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, 0052):  # 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(1,1000):  #Stick some quiet space after
that
setSampleValueAt(target, targetIndex,0)
targetIndex = targetIndex + 1

play(target)         #Let's hear and return the result
return target
```

We'd use it as simply as saying:

```
>>> newSound=splicePreamble()
```

There's a lot going on in this recipe! Let's walk through it, slowly.

Notice that there are lots of lines with “#” in them. The hash character signifies that what comes after that character on the line is a note to the programmer *and should be ignored by Python!* It’s called a *comment*. Comments are great ways to explain what you’re doing to others—and to yourself! The reality is that it’s hard to remember all the details of a program, so it’s often *very* useful to leave notes about what you did if you’ll ever play with the program again.

The function `splice` takes no parameters. Sure, it would be great to write a single function that can do any kind of splicing we want, in the same way as we’ve done generalized increasing volume and normalization. But how could you? How do you generalize all the start and end points? It’s easier, at least to start, to create single recipes that handle specific splicing tasks.

We see here three of those copying loops like we set up earlier. Actually, there are only two. The first one copies the word “United” into place. The second one copies the word “people” into place. “But wait,” you might be thinking. “The word ‘people’ was *already* in the sound!” That’s true, but when we copy “United” in, we overwrite some of the word “people,” so we copy it in again.

Here’s the simpler form. Try it and listen to the result:

```
def spliceSimpler():
file = "/Users/guzdial/mediasources/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
play(target)         #Let’s hear and return the result
return target
```

Let’s see if we can figure out what’s going on mathematically. Recall the table back on page 79. We’re going to start inserting samples at sample index 17408. The word “United” has $(40052 - 33414)$ 6638 samples. (Exercise for the reader: How long is that in seconds?) That means that we’ll be writing into the target from 17408 to $(17408 + 6638)$ sample index 24046. We know from the table that the word “People” ends at index 26726. If the word

"People" is more than (26726 – 24046) 2,680 samples, then it will start earlier than 24046, and our insertion of "United" is going to trample on part of it. If the word "United" is over 6000 samples, I doubt that the word "People" is less than 2000. That's why it sounds crunched. Why does it work with where the "of" is? The speaker must have paused in there. If you check the table again, you'll see that the word "of" ends at sample index 32131 and the word before it ends at 26726. The word "of" takes fewer than (32131 – 26726) 5405 samples, which is why the original recipe works.

The third loop in the original Recipe 14 (page 81) looks like the same kind of copy loop, but it's really only putting in a few 0's. As you might have already guessed, samples with 0's are silent. Putting a few in creates a pause that sounds better. (There's an exercise which suggests pulling it out and seeing what you hear.)

Finally, at the very end of the recipe, there's a new statement we haven't seen yet: `return`. We've now seen many functions in Python that return values. This is how one does it. It's important for `splice` to return the newly spliced sound. Because of the scope of the function `splice`, if the new sound wasn't created, it would simply disappear when the function ended. By returning it, it's possible to give it a name and play it (and even further manipulate it) after the function stops executing.

Figure 3.20 shows the original `preamble10.wav` file in the top sound editor, and the new spliced one (saved with `writeSoundTo`) on the bottom. The lines are drawn so that the spliced section lies between them, while the rest of the sounds are identical.

3.2.6 Backwards sounds

In the splicing example, we copied the samples from the words just as they were in the original sound. We don't have to do always go in the same order. We can reverse the words—or make them faster, slower, louder, or softer. For an example, here's a recipe that plays a sound in a file, backwards.

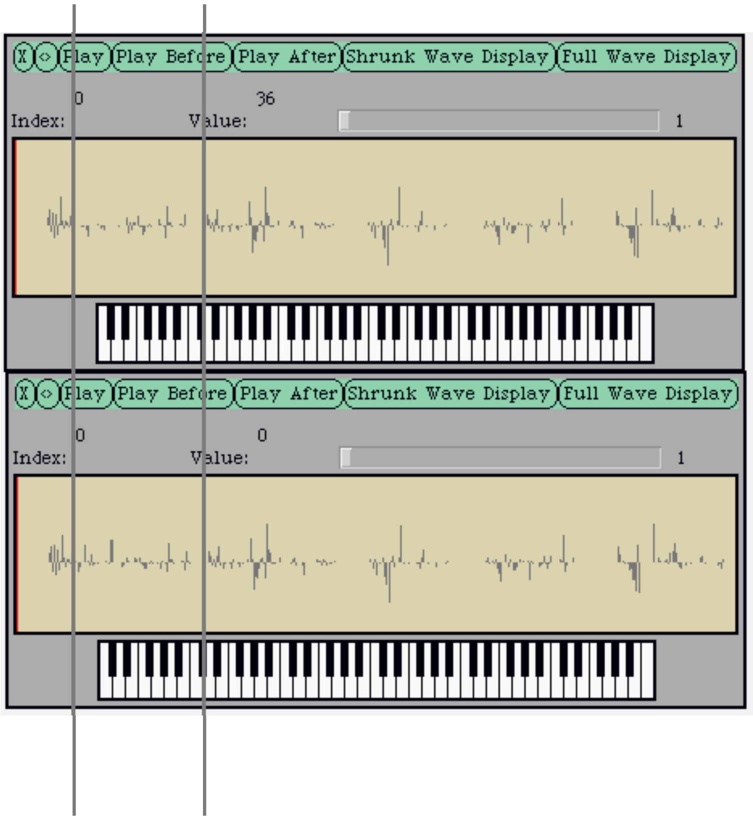


Figure 3.20: Comparing the original sound (top) to the spliced sound (bottom)

Recipe 15: Play the given sound backwards

```
def backwards(filename):
  source = makeSound(filename)
  target = makeSound(filename)

  sourceIndex = getLength(source)
  for targetIndex in range(1,getLength(target)+1):
    sourceValue = getSampleValueAt(source,sourceIndex)
    setSampleValueAt(target,targetIndex,sourceValue)
    sourceIndex = sourceIndex - 1

  return target
```

This recipe uses another variant of the array element copying sub-recipe that we've seen previously.

- The recipe starts the `sourceIndex` at the *end* of the array, rather than the front.
- The `targetIndex` moves from 1 to the length, during which time the recipe:
  - Get the sample value in the source at the `sourceIndex`.
  - Copy that value into the target at the `targetIndex`
  - *Reduce* the `sourceIndex` by 1, meaning that the `sourceIndex` moves from the end of the array back to the beginning.

3.2.7 Changing the frequency of sounds

Modern music and sound keyboards (and synthesizers) allow musicians to record sounds in their daily lives, and turn them into "instruments" by shifting the frequency of the original sounds. How do the synthesizers do it? It's not really complicated, but it certainly is non-intuitive—at first. The interesting part is that it allows you to use any sound you want and make it into an instrument.

This first recipe works by creating a sound that *skips* every other sample. You read that right. After being so careful treating all the samples the same, we're now going to skip half of them! In the `mediasources` directory, you'll find a sound named `c4.wav`. This is the note C, in the fourth octave of a piano, played for one second. It makes a good sound to experiment with, though really, any sound will work.

Recipe 16: Double the frequency of a sound

```
def double(filename):
  source = makeSound(filename)
  target = makeSound(filename)

  targetIndex = 1
  for sourceIndex in range(1, getLength(source)+1, 2):
    setSampleValueAt( target, targetIndex,
getSampleValueAt( source, sourceIndex))
    targetIndex = targetIndex + 1

  #Clear out the rest of the target sound -- it's only half
full!
  for secondHalf in range( getLength( target)/2, getLength(
target)):
    setSampleValueAt(target,targetIndex,0)
    targetIndex = targetIndex + 1

  play(target)
  return target
```

Here's how I use it.

```
>>> file = pickAFile()
>>> print file
/Users/guzdial/mediasources/c4.wav
>>> c4 = makeSound(file)
>>> play(c4)
>>> c4doubled=double(file)
```

This recipe looks like it's using the array-copying sub-recipe we saw earlier, but notice that the `range` uses the third parameter—we're incrementing by two. If we increment by two, we only fill half the samples in the target, so the second loop just fills the rest with zeroes.

Try it[4]! You'll see that the sound really does double in frequency!

[4] You are now trying this out as you read, aren't you?

How did that happen? It's not really all that complicated. Think of it this way. The frequency of the basic file is really the number of cycles that pass by in a certain amount of time. If you skip every other sample, the new sound has just as many cycles, but has them in half the amount of time!

Now let's try the other way: Let's take every sample twice! What happens then?

To do this, we need to learn a new Python function: `int`. `int` returns the integer portion of the input.

```
>>> print int(0.5)
0
>>> print int(1.5)
1
```

Here's the recipe that *halves* the frequency. We're using the array-copying sub-recipe again, but we're sort of reversing it. The `for` loop moves the `targetIndex` along the length of the sound. The `sourceIndex` is now being incremented–but only by 0.5! The effect is that we'll take every sample in the source twice. The `sourceIndex` will be 1, 1.5, 2, 2.5, and so on, but because we're using the `int` of that value, we'll take samples 1, 1, 2, 2, and so on.

Recipe 17: Half the frequency

```
def half(filename):
  source = makeSound(filename)
  target = makeSound(filename)

  sourceIndex = 1
  for targetIndex in range(1, getLength( target)+1):
    setSampleValueAt( target, targetIndex,
getSampleValueAt( source, int(sourceIndex)))
    sourceIndex = sourceIndex + 0.5

  play(target)
  return target
```

Think about what we're doing here. Imagine that the number 0.5 above were actually 0.75. Or 2. or 3. Would this work? The `for` loop would have to change, but essentially the idea is the same in all these cases. We are *sampling* the source data to create the target data. Using a *sample index* of 0.5 slows down the sound and halves the frequency. A sample index larger than one speeds up the sound and increases the frequency.

Let's try to generalize this sampling with the below recipe. (Note that this one *won't* work right!)

Recipe 18: Shifting the frequency of a sound: BROKEN!

```
def shift(filename,factor):
  source = makeSound(filename)
  target = makeSound(filename)

  sourceIndex = 1
  for targetIndex in range(1, getLength( target)+1):
    setSampleValueAt( target, targetIndex,
getSampleValueAt( source, int(sourceIndex)))
    sourceIndex = sourceIndex + factor

  play(target)
  return target
```

Here's how we could use this:

```
>>> hello=pickAFile()
>>> print hello
/Users/guzdial/mediasources/hello.wav
>>> lowerhello=shift(hello,0.75)
```

That will work really well! But what if the `factor` for sampling is *MORE* than 1.0?

```
>>> higherhello=shift(hello,1.5)
I wasn't able to do what you wanted.
The error java.lang.ArrayIndexOutOfBoundsException has occured
Please check line 7 of /Users/guzdial/shift-broken.py
```

Why? What's happening? Here's how you could see it: Print out the `sourceIndex` just before the `setSampleValueAt`. You'd see that the `sourceIndex` becomes *larger* than the source sound! Of course, that makes sense. If each time through the loop, we increment the `targetIndex` by 1, but we're incrementing the `sourceIndex` by *more than one*, we'll get past the end of the source sound before we reach the end of the target sound. But how do we avoid it?

Here's what we want to happen: If the `sourceIndex` ever gets larger than length of the source, we want to reset the sourceIndex—probably back to 1. The key word there is *if*, or even `if`. It turns out that we can *can* tell Python to make decisions based on a *test* and do something based on *if* something is true. In our case, the test is `sourceIndex > getLength(source)`. We can test on `<`, `>`, `==` (for equality), and even `<=` and `>=`. An `if` statement takes a *block*, just as `def` and `for` do. The block defines the things to do if the *test* in the `if` statement is true. In this case, our block is simply `sourceIndex = 1`.

The below recipe generalizes this and allows you to specify how much to shift the samples by.

Recipe 19: Shifting the frequency of a sound

```
def shift(filename,factor):
  source = makeSound(filename)
  target = makeSound(filename)

  sourceIndex = 1
  for targetIndex in range(1, getLength( target)+1):
    setSampleValueAt( target, targetIndex,
getSampleValueAt( source, int(sourceIndex)))
    sourceIndex = sourceIndex + factor
    if sourceIndex > getLength(source):
      sourceIndex = 1

  play(target)
  return target
```

We can actually set the factor so that we get whatever frequency we want. We call this factor the *sampling interval*. For a desired frequency f_0, the sampling interval should be:

$(sizeofsourcesound)\frac{f_0}{samplingrate}$

This is how a keyboard synthesizer works. It has recordings of pianos, voices, bells, drums, whatever. By *sampling* those sounds at different sampling intervals, it can shift the sound to the desired frequency.

Functions and Objects Summary

In this chapter, we talk about several kinds of encodings of data (or objects).

Sounds	Sounds are encodings of sounds, typically coming froma WAV file.
Samples	Samples are collections of Sample objects, each indexed by a number, e.g. sample #1, sample #2, etc. `samples[1]` is the first Sample object. You can manipulate each Sample in the Samples like this `for s in samples:`.
Sample	A sample is a value between -32000 and 32000 (roughly) representing the voltage that a microphone would generate at a given instant when recording a sound. The length of the instant is typically either $1/44,1000$ of a second (for CD-quality sound) or $1/22,050$ of a second (for good enough sound on most computers). A Sample object remembers what sound it came from, so if you change its value, it knows to go back and change the right sample in the sound.

Here are the functions used or introduced in this chapter:

range	Takes two numbers, and returns an array of all integers starting at the first number and stopping before the last number.
range	Can also take three numbers, and then returns an array of all the integers from the first, up-to-but-not-including the second, incrementing each time by the third.
int	Returns the integer part of the input value
max	Takes as many numbers as you want, and returns the largest value

pickAFile	Lets the user pick a file and returns the complete path name as a string. No input
makeSound	Takes a filename as input, reads the file, and creates a sound from it. Returns the sound.
play	Plays a sound provided as input. No return value.
getLength	Takes a sound as input and returns the number of samples in that sound.
getSamples	Takes a sound as input and returns the Samples in that sound.
blockingPlay	Plays the sound provided as input, and makes sure that no other sound plays at the exact same time. (Try two `play`'s right after each other.)
playAtRate	Takes a sound and a rate (1.0 means normal speed, 2.0 is twice as fast, and 0.5 is half as fast), and plays the sound at that rate. The duration is always *the same*, e.g., if you play it twice as fast, the sound plays *twice* to fill the given time.
playAtRateDur	Takes a sound, a rate, and a duration as the number of samples to play.
writeSoundTo	Takes a sound and a filename (a string) and writes the sound to that file as a WAV file. (Make sure that the filename ends in ".wav" if you want the operating system to treat it right.)
getSamplingRate	Takes a sound as input and returns the number representing the number of samples in each second for the sound.
getLength	Returns the length of the sound as a number of samples
getSampleValueAt	Takes a sound and an index (an integer value), and returns the value of the sample (between -32000 and 32000) for that object.
setSampleValueAt	Takes a sound, an index, and a value (should be between -32000 and 32000), and sets the value of the sample at the given index in the given sound to the given value.
getSampleObjectAt	Takes a sound and an index (an integer value), and returns the Sample object at that index.

getSample	Takes a Sample object and returns its value (between -32000 and 32000)
setSample	Takes a Sample object and a value, and sets the sample to that value.
getSound	Takes a Sample object and returns the Sound that it remembers as its own.

Exercises

Exercise 13: Open up the SONOGRAM view and say some vowel sounds. Is there a distinctive pattern? Do "Oh's" always sound the same? Do "Ah's"? Does it matter if you switch speakers—are the patterns the same?

Exercise 14: Get a couple of different instruments and play the same note on them into MediaTools sound editor with the sonogram view open. Are all "C's" made equal? Can you see some of why one sound is different than another?

Exercise 15: Try out a variety of WAV files as instruments, using the piano keyboard in the MediaTools sound editor. What kinds of recordings work best as instruments?

Exercise 16: Recipe 9 (page 66) takes a sound as input. Write a function `increaseVolumeNamed` that takes a file name as input then `play` the louder sound.

Exercise 17: Rewrite Recipe 9 (page 66) so that it takes two inputs: The sound to increase in volume, and a filename where the newly louder sound should be stored. Then, increase the volume, and write the sound out to the name file. Also, try doing it taking an input filename instead of the sound, so that inputs are both filenames.

Exercise 18: Rewrite Recipe 9 (page 66) so that it takes two inputs: A sound to increase in volume, and a *multiplier*. Use the multiplier as *how much* to increase the amplitude of the sound samples. Can we use this same function to both increase and decrease the volume? Demonstrate commands that you would execute to do each.

Exercise 19: In section 3.2.3, we walked through how Recipe 9 (page 66) worked. Draw the pictures to show how Recipe 10 (page 72) works, in the same way.

Exercise 20: What happens if you increase a volume too far? Do it once, and again, and again. Does it always keep getting louder? Or does something else happen? Can you explain why?

Exercise 21: Try sprinkling in some specific values into your sounds.

What happens if you put a few hundred 32767 samples into the middle of a sound? Or a few hundred -32768? Or a bunch of zeroes? What happens to the sound?

Exercise 22: In Recipe 12 (page 77), we add one to `getLength(sound)` in the `range` function. Why'd we do that?

Exercise 23: Rewrite Recipe 13 (page 77) so that two input values are provided to the function: The sound, and a *percentage* of how far into the sound to go before dropping the volume.

Exercise 24: Rewrite Recipe 13 (page 77) so that you normalize the first second of a sound, then slowly decrease the sound in steps of 1/5 for each following second. (How many samples are in a second? `getSamplingRate` is the number of samples per second for the given sound.)

Exercise 25: Try rewriting Recipe 13 (page 77) so that you have a linear increase in volume to halfway through the sound, then linearly decrease the volume down to zero in the second half.

Exercise 26: What happens if you take out the bit of silence added in to the target sound in Recipe 14 (page 81)? Try out? Can you hear any difference?

Exercise 27: I think that if we're going to say "We the UNITED people" in Recipe 14 (page 81), the "UNITED" should be really emphasized—really loud. Change the recipe so that the word "united" is louder in the phrase "united people."

Exercise 28: How long is a sound compared to the original when it's been doubled by Recipe 16 (page 86)?

Exercise 29: Hip-hop DJ's move turntables so that sections of sound are moved forwards and backwards quickly. Try combining Recipe 15 (page 84) and Recipe 16 (page 86) to get the same effect. Play a second of a sound quickly forward, then quickly backward, two or three times. (You might have to move faster than just double the speed.)

Exercise 30: Try using a stopwatch to time the execution of the recipes in this chapter. Time from hitting return on the command, until the next prompt appears. What is the relationship between execution time and the length of the sound? Is it a linear relationship, i.e., longer sounds take longer to process and shorter sounds take less time to process? Or is it something else? Compare the individual recipes. Does normalizing a sound take longer than raising (or lowering) the amplitude a constant amount? How much longer? Does it matter if the sound is longer or shorter?

Exercise 31: Consider changing the `if` block in Recipe 19 (page 89) to `sourceIndex = sourceIndex - getLength(source)`. What's the difference from just setting the `sourceIndex` to 1? Is this better or worse? Why?

Exercise 32: If you use Recipe 19 (page 89) with a factor of 2.0 or 3.0, you'll get the sound repeated or even triplicated. Why? Can you fix it? Write `shiftDur` that takes a number of samples (or even seconds) to play the sound.

Exercise 33: Change the `shift` function in Recipe 19 (page 89) to `shiftFreq` which takes a frequency instead of a factor, then plays the given sound at the desired frequency.

To Dig Deeper

There are many wonderful books on psychoacoustics and computer music. One of my favorites for understandability is *Computer Music: Synthesis, Composition, and Performance* by Dodge and Jerse [Dodge and Jerse, 1997]. The *bible* of computer music is Curtis Roads' massive *The Computer Music Tutorial* [Roads, 1996].

When you are using MediaTools, you are actually using a programming language called *Squeak*, developed initially and primarily by Alan Kay, Dan Ingalls, Ted Kaehler, John Maloney, and Scott Wallace [Ingalls et al., 1997]. Squeak is now open-source[5], and is an excellent cross-platform multimedia tool. There is a good book introducing Squeak, including the sound capabilities [Guzdial, 2001], and another book on Squeak [Guzdial and Rose, 2001] that includes a chapter on *Siren*, an off-shoot of Squeak by Stephen Pope especially designed for computer music exploration and composition.

[5] `http://www.squeak.org`

Chapter 4

Creating Sounds

XXX THIS CHAPTER IS STILL ROUGH

Creating sounds digitally that didn't exist previously is lots of fun. Rather than simply moving around samples or multiplying them, we actually change their values—add waves together. The result are sounds that never existed until you made them.

In physics, adding sounds involves issues of cancelling waves out and enforcing other factors. In math, it's about matrices. In computer science, it's the easiest process in the world! Let's say that you've got a sound, `source`, that you want to add in to the `target`. *Simply add the values at the same index numbers!* That's it!

```
for sourceIndex in range(1,getLength(source)+1):
  targetValue=getSampleValueAt(target,sourceIndex)
  sourceValue=getSampleValueAt(source,sourceIndex)
  setSampleValueAt(source,sourceIndex,sourceValue+targetValue)
```

To make some of our manipulations easier, we're going to start using a shorthand for accessing media files. JES knows how *set* a media folder, and then reference media files within that folder. This makes it much easier to reference media files—you don't have to spell out the whole path. The functions we'll use are `setMediaFolder` and `getMediaPath`. `setMediaFolder` will put up a file picker—pick any file in your media folder. `getMediaPath` takes a base file name as an argument, and will stick the path to the media folder in front of the base name and return a whole path to it.

```
>>> setMediaFolder()
New media folder: /Users/guzdial/mediasources/
```

```
>>> print getMediaPath("barbara.jpg")

/Users/guzdial/mediasources/barbara.jpg

>>> print getMediaPath("sec1silence.wav")

/Users/guzdial/mediasources/sec1silence.wav
```

Common Bug: It's not a file, it's a string
Just because `getMediaPath` returns something that looks like a path doesn't mean that a file really exists there. You have to know the right base name, but if you do, it's easier to use in your code. But if you put in a non-existent file, you'll get a path to a non-existent file. `getMediaPath` will warn you.

```
>>> print getMediaPath("blah-blah-blah")
Note: There is no file at
/Users/guzdial/mediasources/blah-blah-blah
/Users/guzdial/mediasources/blah-blah-blah
```

4.1 Creating an Echo

Creating an echo effect is similar to the splicing recipe (Recipe 14 (page 81)) that we saw in the last chapter, but involves actually creating sounds that didn't exist before. We do that by actually *adding* wave forms. What we're doing here is adding samples from a `delay` number of samples away into the sound, but multiplied by 0.6 so that they're fainter.

Recipe 20: Make a sound and a single echo of it

```
def echo(delay):
  f = pickAFile()
  s1 = makeSound(f)
  s2 = makeSound(f)
  for p in range(delay+1, getLength(s1)):
    # set delay to original value + delayed value * .6
    setSampleValueAt(s1, p, getSampleValueAt( s1,p) +
.6*getSampleValueAt( s2, p-delay) )

  play(s1)
```

4.1.1 Creating Multiple Echoes

This recipe actually lets you set the number of echoes that you get. You can generate some amazing effects like this.

Recipe 21: Creating multiple echoes

```
def echoes(delay,echoes):
  f = pickAFile()
  s1 = makeSound(f)
  s2 = makeSound(f)
  endCurrentSound = getLength(s1)
  newLength = endCurrentSound+(echoes * delay)      # get
ultimate length of sound

  for i in range (endCurrentSound,newLength+1):
    # initialize delay samples to zero
    setSampleValueAt(s1,i,0)

  echoAmplitude = 1
  for echoCount in range (1, echoes+1):  # for each echo
    echoAmplitude *= .6                     # decrement
amplitude  to .6 of current volume
    for e in range (1,endCurrentSound+1):      # loop
through the entire sound
      position = e+delay*echoCount            # increment
position by one
            # Set this sample's value to the original value
plus the amplitude * the original sample value
      setSampleValueAt(s1,position, getSampleValueAt(s1,
position) + echoAmplitude * getSampleValueAt( s2,
position-(delay*echoCount) ) )
  play(s1)
```

4.1.2 Additive Synthesis

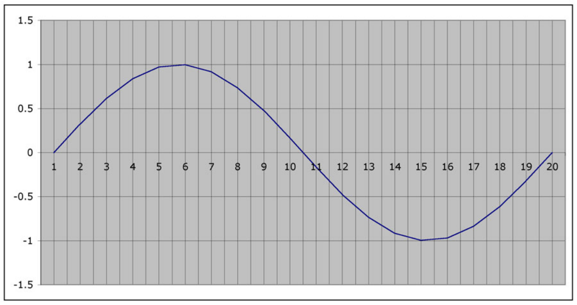
Additive synthesis creates sounds by adding sine waves together. We saw earlier that it's really pretty easy to add sounds together. With additive synthesis, you can shape the waves yourselves, set their frequencies, and create "instruments" that have never existed.

Making sine waves

Let's figure out how to produce a set of samples to generate a sound at a given frequency and amplitude.

From trignometry, we know that if we take the sine of the radians from 0 to 2π, we'll get a circle. Spread that over time, and you get a sine wave. In other words, if you took values from 0 to 2Π, computed the sine of each value, and graphed the computed values. You'd get a sine wave. From your really early math courses, you know that there's an infinity of numbers between 0 and 1. Computers don't handle infinity very well, so we'll actually only take *some* values between 0 to 2Π.

To create the below graph, I filled 20 rows (a totally arbitrary number) of a spreadsheet with values from 0 and 2Π (about 6.28). I added about 0.314 (6.28/20) to each preceeding row. In the next column, I took the sine of each value in the first column, then graphed it.



Now, if we want to create a sound at a given frequency, say 440 Hz. This means that we have to fit an entire cycle like the above into 1/440 of a second. (440 cycles per second, means that each cycle fits into 1/440 second, or 0.00227 seconds.) I made the above picture using 20 values. Call it 20 *samples*. How many samples to I have to chop up the 440 Hz cycle into? That's the same question as: How many samples must go by in 0.00227 seconds? We know the sampling rate—that's the number of samples in one second. Let's say that it's 22050 samples per second (our default sampling rate). Each sample is then (1/22050) 0.0000453 seconds. How many samples fit into 0.00227? That's 0.00227/0.0000453, or about 50. What we just did here mathematically is:

$interval = 1/frequency$

$samplesPerCyle = \frac{interval}{1/samplingRate} = (samplingRate)(interval)$

Now, let's spell this out as Python. To get a waveform at a given frequency, say 440 Hz, we need 440 of these waves in a single second. Each one must fit into the interval of $1/frequency$. The number of samples that needs to be produced during the interval is the sampling rate divided by the frequency, or interval $(1/f) * (samplingrate)$. Call that the $samplesPerCycle$.

At each entry of the sound $sampleIndex$, we want to:

- Get the fraction of $sampleIndex/samplesPerCycle$.

- Multiply that fraction by 2Π. That's the number of radians we need. Take the *sin* of $(sampleIndex/samplesPerCycle) * 2\Pi$.

- Multiply the result by the desired amplitude, and put that in the `sampleIndex`.

To build sounds, there are some *silent* sounds in the media sources. Our sine wave generator will use one second of silence to build a sine wave of one second. We'll provide an amplitude as input—that will be the *maximum* amplitude of the sound. (Since sine generates between -1 and 1, the range of amplitudes will be between $-amplitude$ and $amplitude$.)

Common Bug: Set the media folder first!
If you're to use code that uses `getMediaPath`, you'll need to execute `setMediaFolder` first.

Recipe 22: Generate a sine wave at a given frequency and amplitude

```
def sineWave(freq,amplitude):

    # Get a blank sound
    mySound =  getMediaPath('sec1silence.wav')
    buildSin = makeSound(mySound)

    # Set sound constant
    sr = getSamplingRate(buildSin)        # sampling rate

    interval = 1.0/freq      # Make sure it's floating
point
    samplesPerCycle =  interval * sr    # samples per cycle
    maxCycle = 2 * pi

    for pos in range (1,getLength(buildSin)+1):
        rawSample = sin( (pos / samplesPerCycle) *
maxCycle)
        sampleVal = int( amplitude*rawSample)
        setSampleValueAt(buildSin,pos,sampleVal)

    return (buildSin)
```

Here we are building a sine wave of 880 Hz at an amplitude of 4000.

```
>>> f880=sineWave(880,4000)
>>> play(f880)
```

Adding sine waves together

Now, let's add sine waves together. Like we said at the beginning of the chapter, that's pretty easy: Just add the samples at the same indices together. Here's a function that adds one sound into a second sound.

Recipe 23: Add two sounds together

```
def addSounds(sound1,sound2):
  for index in range(1,getLength(sound1)+1):
    s1Sample = getSampleValueAt(sound1,index)
    s2Sample = getSampleValueAt(sound2,index)
    setSampleValueAt(sound2,index,s1Sample+s2Sample)
```

How are we going to use this function to add together sine waves? We need both of them at once? Turns out that it's easy:

Making it Work Tip: You can put more than one function in the same file!
It's perfectly okay to have more than one function in the same file. Just type them all in in any order. Python will figure it out.

My file `additive.py` looks like this:

```
def sineWave(freq,amplitude):

    # Get a blank sound

    mySound =  getMediaPath('sec1silence.wav')
    buildSin = makeSound(mySound)

    # Set sound constant
    sr = getSamplingRate(buildSin)        # sampling rate

    interval = 1.0/freq
    samplesPerCycle =  interval * sr    # samples per cycle:
make sure floating point
    maxCycle = 2 * pi
```

```
  for pos in range (1,getLength(buildSin)+1):
    rawSample = sin( (pos / samplesPerCycle) * maxCycle)
    sampleVal = int( amplitude*rawSample)
    setSampleValueAt(buildSin,pos,sampleVal)

  return (buildSin)

def addSounds(sound1,sound2):
 for index in range(1,getLength(sound1)+1):
  s1Sample = getSampleValueAt(sound1,index)
  s2Sample = getSampleValueAt(sound2,index)
  setSampleValueAt(sound2,index,s1Sample+s2Sample)
```

Let's add together 440 Hz, 880 Hz (twice 440), and 1320 Hz (880+440), but we'll have the amplitudes increase. We'll double the amplitude each time: 2000, then 4000, then 8000. We'll add them all up into the name `f440`. At the end, I generate a 440 Hz sound so that I can listen to them both and compare.

```
>>> f440=sineWave(440,2000)
>>> f880=sineWave(880,4000)
>>> f1320=sineWave(1320,8000)
>>> addSounds(f880,f440)
>>> addSounds(f1320,f440)
>>> play(f440)
>>> just440=sineWave(440,2000)
>>> play(just440)
```

Common Bug: Beware of adding amplitudes past 32767
When you add sounds, you add their amplitudes, too. A maximum of 2000+4000+8000 will never be greater than 32767, but do worry about that. Remember what happened when the amplitude got too high last chapter. . .

Checking our result

How do we know if we really got what we wanted? We can test our code by using the sound tools in the MediaTools. First, we save out a sample wave (just 400 Hz) and the combined wave.

```
>>> writeSoundTo(just440,"/Users/guzdial/mediasources/just440.wav")

>>> writeSoundTo(f440,"/Users/guzdial/mediasources/combined440.wav")
```

Open up each of these in turn in the sound editor. Right away, you'll notice that the wave forms look very different (Figure 4.2). That tells you that we did *something* to the sound, but what?

The way you can really check your additive synthesis is with an FFT. Generate the FFT for each signal. You'll see that the 440 Hz signal has a single spike (Figure 4.3). That's what you'd expect—it's supposed to be a single sine wave. Now, look at the combined wave form's FFT (Figure 4.4). Wow! It's what it's supposed to be! You see three spikes there, and each succeeding one is double the height of the last one.

Square waves

We don't have to just add sine waves. We can also add *square waves*. These are literally square-shaped waves, moving between +1 and −1. The FFT will look very different, and the *sound* will be very different. It can actually be a much richer sound.

Try swapping this recipe in for the sine wave generator and see what you think. Note the use of an `if` statement to swap between the positive and negative sides of the wave half-way through a cycle.

Recipe 24: Square wave generator for given frequency and amplitude

```
def squareWave(freq,amplitude):

  # Get a blank sound
  mySound =  getMediaPath("sec1silence.wav")
  square = makeSound(mySound)

  # Set music constants
  samplingRate = getSamplingRate(square)             #
sampling rate
  seconds = 1                             # play for 1
second

  # Build tools for this wave
  interval = 1.0 * seconds / freq               # seconds
per cycle: make sure floating point
  samplesPerCycle = interval * samplingRate     # creates
floating point since interval is fl point
  samplesPerHalfCycle = int(samplesPerCycle / 2) # we need
to switch every half-cycle
  sampleVal = amplitude
  s = 1
  i = 1

  for s in range (1, getLength(square)+1):    # create 1
second sound
    if (i > samplesPerHalfCycle):   # if end of a
half-cycle
      sampleVal = sampleVal * -1      # reverse the
amplitude every half-cycle
      i = 0                           # and reinitialize
the half-cycle counter
    setSampleValueAt(square,s,sampleVal)
    i = i + 1

  return(square)
```

Use it like this:

```
>>> sq440=squareWave(440,4000)
>>> play(sq440)
>>> sq880=squareWave(880,8000)
>>> sq1320=squareWave(1320,10000)
>>> writeSoundTo(sq440,getMediaPath("square440.wav"))
Note: There is no file at /Users/guzdial/mediasources/square440.wav
>>> addSounds(sq880,sq440)
>>> addSounds(sq1320,sq440)
>>> play(sq440)
>>> writeSoundTo(sq440,getMediaPath("squarecombined440.wav"))
Note: There is no file at /Users/guzdial/mediasources/squarecombined440.wav
```

You'll find that the waves (in the wave editor of MediaTools) really do look square (Figure 4.5), but the most amazing thing is all the additional spikes in FFT (Figure 4.6). Square waves really do result in a much more complex sound.

Exercises

Exercise 34: Using the sound tools, figure out the characteristic pattern of different instruments. For example, pianos tend to have a pattern the opposite of what we created—the amplitudes *decrease* as we get to higher sine waves. Try creating a variety of patterns and see how they sound and how they look.

Exercise 35: When musicians work with additive synthesis, they will often wrap *envelopes* around the sounds, and even around each added sine wave. An envelope *changes* the amplitude over time: It might start out small, then grow (rapidly or slowly), then hold at a certain value during the sound, and then drop before the sound ends. That kind of pattern is sometimes called the *attack-sustain-decay (ASD) envelope*. Try implementing that for the sine and square wave generators.

To Dig Deeper

Good books on computer music will talk a lot about creating sounds from scratch like in this chapter. One of my favorites for understandability is *Computer Music: Synthesis, Composition, and Performance* by Dodge and Jerse [Dodge and Jerse, 1997]. The *bible* of computer music is Curtis Roads' massive *The Computer Music Tutorial* [Roads, 1996].

One of the most powerful tools for playing with this level of computer music is *CSound*. It's a software music synthesis system, free, and totally cross-platform. The book by Richard Boulanger [Boulanger, 2000] has everything you need for playing with CSound.

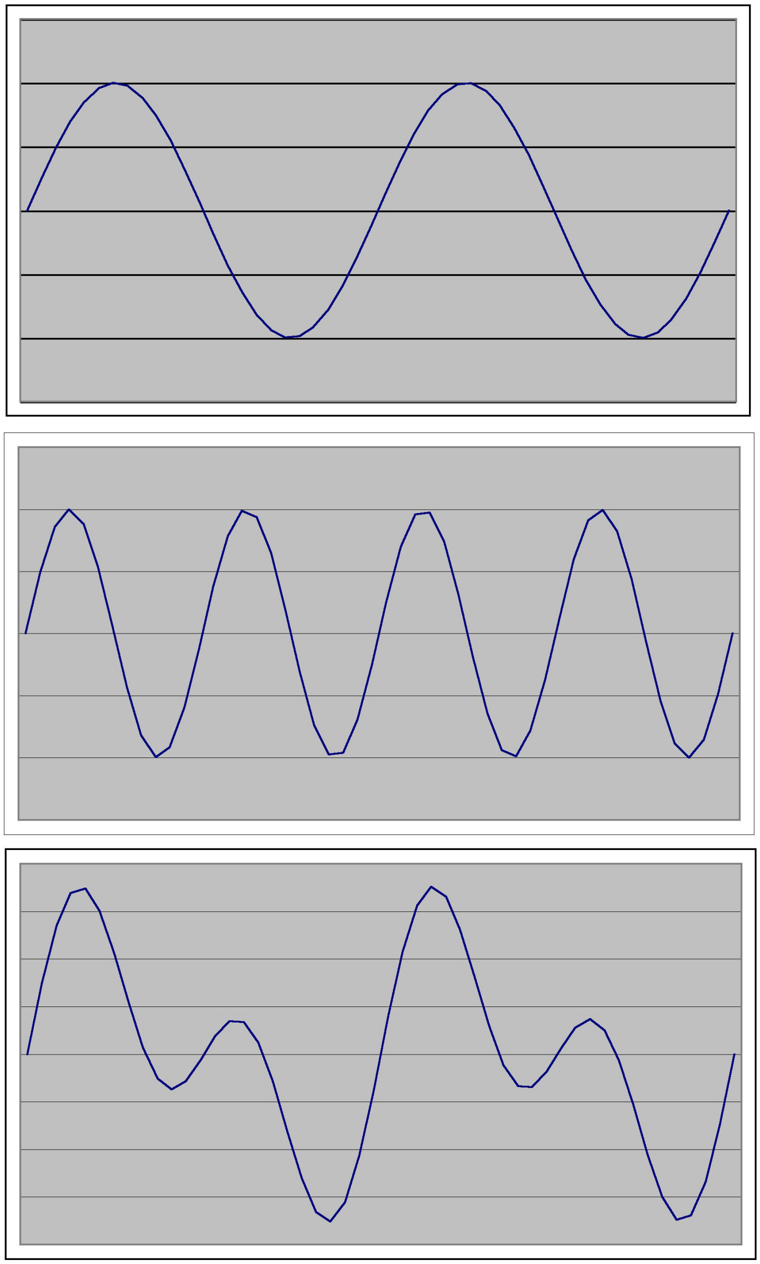


Figure 4.1: The top and middle waves are added together to create the bottom wave

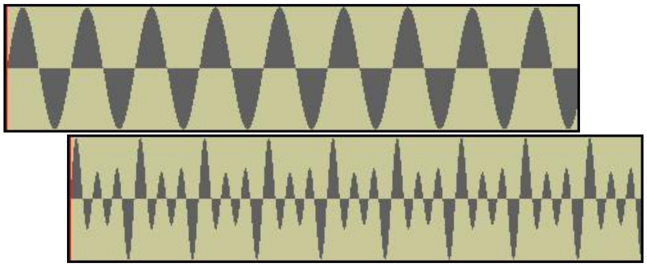


Figure 4.2: The raw 440 Hz signal on top, then the 440+880+1320 Hz signal on the bottom

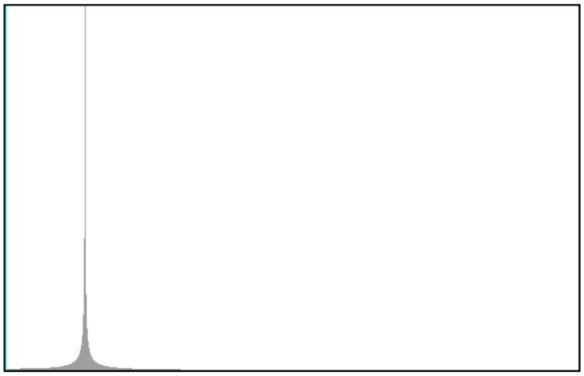


Figure 4.3: FFT of the 440 Hz sound

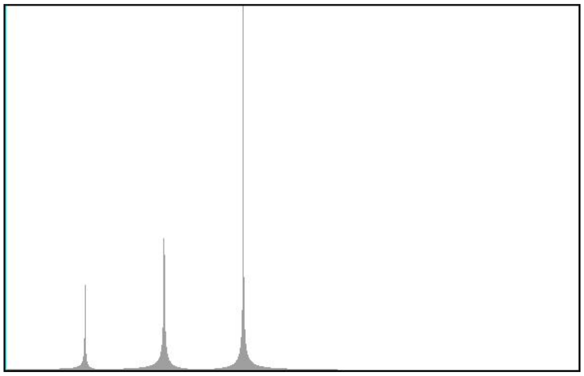


Figure 4.4: FFT of the combined sound

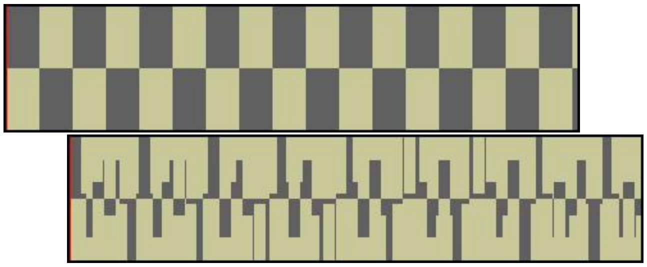


Figure 4.5: The 440 Hz square wave (top) and additive combination of square waves (bottom)

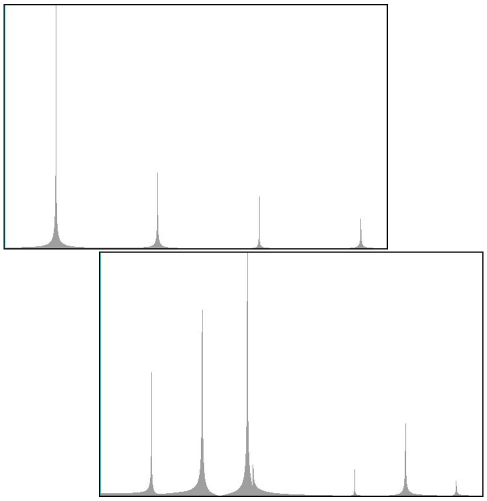


Figure 4.6: FFT's of the 440 Hz square wave (top) and additive combination of square waves (bottom)

Part III

Pictures

Chapter 5

Encoding and Manipulating Pictures

Pictures (images, graphics) are an important part of any media communication. In this chapter, we discuss how pictures are represented on a computer (mostly as *bitmap* images—each dot or *pixel* is represented separately) and how they can be manipulated. The next chapter will discuss more about other kinds of representations, such as *vector* images.

5.1 How Pictures are Encoded

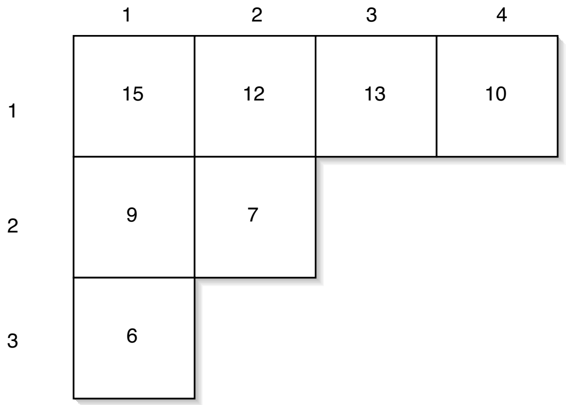
Pictures are two-dimensional arrays of *pixels*. In this section, each of those terms will be described.

For our purposes, a picture is an image stored in a JPEG file. JPEG is an international standard for how to store images with high quality but in little space. JPEG is a *lossy compression* format. That means that it is *compressed*, made smaller, but not with 100% of the quality of the original format. Typically, though, what gets thrown away is stuff that you don't see or don't notice anyway. For most purposes, a JPEG image works fine.

A two-dimensional array is a *matrix*. Recall that we described an array as a sequence of elements, each with an index number associated with it. A matrix is a collection of elements arranged in both a horizontal and vertical sequence. Instead of talking about element at index j, that is $array_j$, we're now talking about element at column i and row$j$, that is, $matrix_{i,j}$.

In Figure 5.1, you see an example matrix (or part of one, the upper-left-hand corner of one). At *coordinates* $(1, 2)$ (horizontal, vertical), you'll find the matrix element whose value is 9. $(1, 1)$ is 15, $(2, 1)$ is 12, and $(3, 1)$ is

13. We will often refer to these coordinates as (x, y) $((horizontal, vertical)$.



	1	2	3	4
1	15	12	13	10
2	9	7		
3	6			

Figure 5.1: An example matrix

What's stored at each element in the picture is a *pixel*. The word "pixel" is short for "picture element." It's literally a dot, and the overall picture is made up of lots of these dots. Have you ever taken a magnifying glass to pictures in the newspaper or magazines, or to a television or even your own monitor? (Figure 5.2 was generated by taking an Intel microscope and pointing it at the screen at $60x$ magnification.) It's made up of many, many dots. When you look at the picture in the magazine or on the television, it doesn't look like it's broken up into millions of discrete spots, but it is.

Just like the samples that make up a sound, our human sensor apparatus can't distinguish (without magnification or other special equipment) the small bits in the whole. That's what makes it possible to digitize pictures. We break up the picture into smaller elements (pixels), but enough of them that the picture doesn't look choppy when looked at it overall. If you *can* see the effects of the digitization (e.g., lines have sharp edges, you see little rectangles in some spots), we call that *pixelization*—the effect when the digitization process becomes obvious.

Picture encoding is only the next step in complexity after sound encoding. A sound is inherently linear—it progresses forward in time. A picture has two dimensions, a width and a height. But other than that, it's quite similar.

We will encode each pixel as a triplet of numbers. The first number represents the amount of red in the pixel. The second is the amount of green, and the third is the amount of blue. It turns out that we can actually

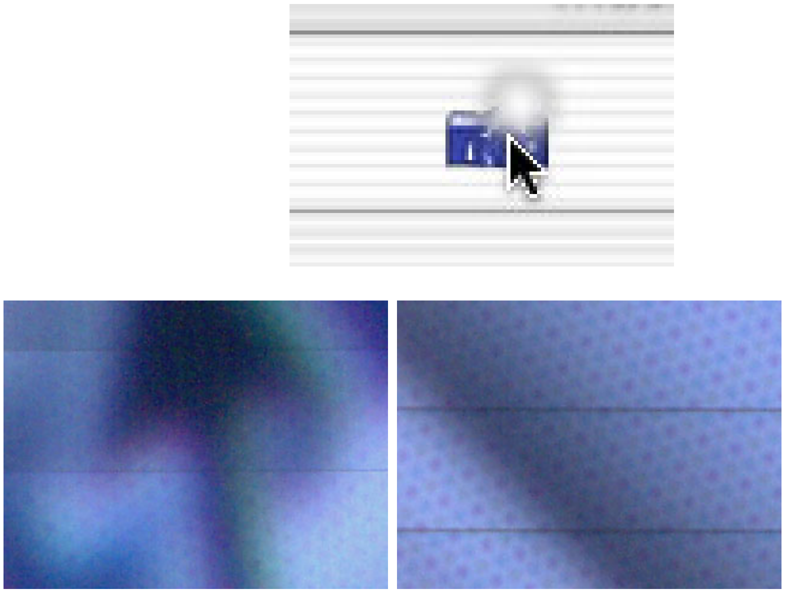


Figure 5.2: Cursor and icon at regular magnification on top, and close-up views of the cursor (left) and the line below the cursor (right)

make up any color by combining red, green, and blue light (Figure 5.3). Combining all three gives us pure white. Turning off all three gives us black. We call this the *RGB model*.

There are other models for defining and encoding colors besides the RGB color model. There's the *HSV color model* which encodes Hue, Saturation, and Value. The nice thing about the HSV model is that some notions, like making a color "lighter" or "darker" map cleanly to it (e.g., you simply change the saturation). Another model is the *CMYK color model*, which encodes Cyan, Magenta, Yellow, and blacK ("B" could be confused with Blue). The CMYK model is what printers use—those are the inks they combine to make colors. However, the four elements means more to encode on a computer, so it's less popular for media computation. RGB is probably the most popular model on computers.

Each color component in a pixel is typically represented with a single byte, eight bits. If you recall our earlier discussion, eight bits can represent 256 values (2^8), which we typically use to represent the values 0 to 255. Each pixel, then, uses 24 bits to represent colors. Using our same formula ($2^2$4), we know that the standard encoding for color using the RGB model can represent 16,777,216 colors. There are certainly more than 16 million

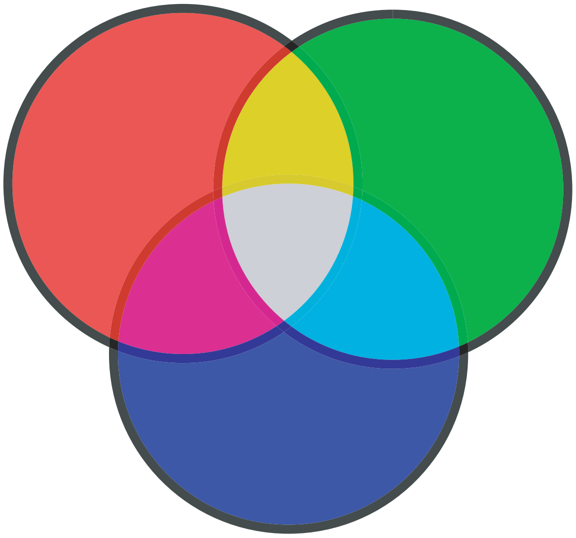
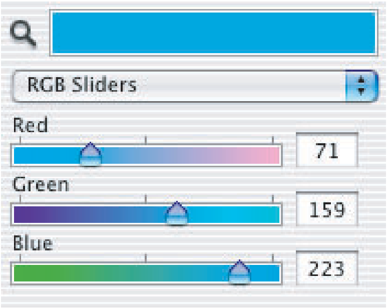


Figure 5.3: Merging red, green, and blue to make new colors

colors in all of creation, but it would take a very discerning eye to pick out any missing in this model.

Most facilities for allowing users to pick out colors let the users specify the color as RGB components. The Macintosh offers RGB sliders in its basic color picker (Figure 5.4). The color chooser in JES (which is the standard Java Swing color chooser) offers a similar set of sliders (Figure 5.5).



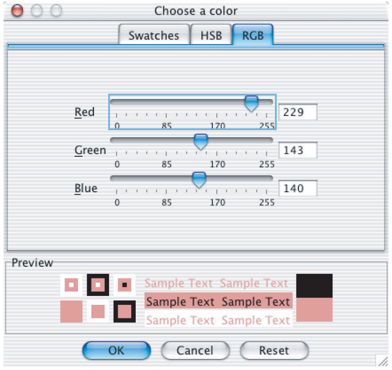
Red 71

Green 159

Blue 223

Figure 5.4: The Macintosh OS X RGB color picker

As mentioned a triplet of $(0, 0, 0)$ (red, green, blue components) is black, and $(0, 0, 0)$ is white. $(255, 0, 0)$ is pure red, but $(100, 0, 0)$ is red, too—just



Red 229
Green 143
Blue 140

Figure 5.5: Picking a color using RGB sliders from JES

less intense. $(0, 100, 0)$ is a light green, and $(0, 0, 100)$ is light blue.

When the red component is the same as the green and as the blue, the resultant color is gray. $(50, 50, 50)$ would be a fairly light gray, and $(100, 100, 100)$ is darker.

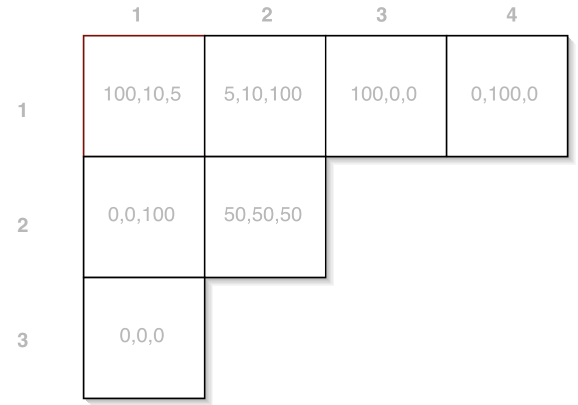
The Figure 5.6 (replicated at Figure 5.27 (page 156) in the color pages at the end of this chapter) is a representation of pixel RGB triplets in a matrix representation. Thus, the pixel at $(2, 1)$ has color $(5, 10, 100)$ which means that it has a red value of 5, a green value of 10, and a blue value of 100—it's a mostly blue color, but not pure blue. Pixel at $(4, 1)$ has a pure green color ($(0, 100, 0)$), but only 100 (out of a possible 255), so it's a fairly light green.

5.2 Manipulating Pictures

We manipulate pictures in JES by making a picture object out of a JPEG file, then changing the pixels in that picture. We change the pixels by changing the color associated with the pixel—by manipulating the red, gree, and blue components. Manipulating pictures, thus, is pretty similar to manipulating samples in a sound, but a little more complex since it's in two dimensions rather than one.

We make pictures using `makePicture`. We make the picture appear with `show`.

```
>>> file=pickAFile()
>>> print file
```



	1	2	3	4
1	100,10,5	5,10,100	100,0,0	0,100,0
2	0,0,100	50,50,50		
3	0,0,0			

Figure 5.6: RGB triplets in a matrix representation

```
/Users/guzdial/mediasources/barbara.jpg
>>> picture=makePicture(file)
>>> show(picture)
>>> print picture
Picture, filename /Users/guzdial/mediasources/barbara.jpg height
294 width 222
```

Pictures know their width and their height. You can query them with `getWidth` and `getHeight`.

```
>>> print getWidth(picture)
222
>>> print getHeight(picture)
294
```

We can get any particular pixel from a picture using `getPixel` with the picture, and the coordinates of the pixel desired. We can also get all the pixels with `getPixels`.

```
>>> pixel=getPixel(picture,1,1)
>>> print pixel
Pixel, color=color r=168 g=131 b=105
>>> pixels=getPixels(picture)
>>> print pixels[0]
Pixel, color=color r=168 g=131 b=105
```

Common Bug: Don't try printing the pixels: *Way* too big!
getPixels literally returns an array of all the pixels (as opposed to a samples object, like getSamples returns). If you try to print the return value from getPixels, you'll get the printout of each pixel, like you see above. How many pixels are there? Well, this small sample picture has a width of 222 and a height of 294. $222x294 = 65,268$ 65 thousand lines like the above is a big printout. You probably don't want to wait for it to finish. If you do this accidentally, just quit JES and re-start it.

Pixels know where they came from. You can ask them their x and y coordinates with getX and getY.

```
>>> print getX(pixel)
1
>>> print getY(pixel)
1
```

Each pixel knows how to getRed and setRed. (Green and blue work similarly.)

```
>>> print getRed(pixel)
168
>>> setRed(pixel,255)
>>> print getRed(pixel)
255
```

You can also ask a pixel for its color with getColor, and you can also set the color with setColor. Color objects know their red, green, and blue components. You can also make new colors with makeColor.

```
>>> color=getColor(pixel)
>>> print color
color r=255 g=131 b=105
>>> setColor(pixel,color)
>>> newColor=makeColor(0,100,0)
>>> print newColor
```

```
color r=0 g=100 b=0
>>> setColor(pixel,newColor)
>>> print getColor(pixel)
color r=0 g=100 b=0
```

If you change the color of a pixel, the picture that the pixel is from does get changed.

```
>>> print getPixel(picture,1,1)
Pixel, color=color r=0 g=100 b=0
```

Common Bug: Seeing changes in the picture If you show your picture, and then change the pixels, you might be wondering, "Where are the changes?!?" Picture displays don't automatically updated. If you execute `repaint` with the picture, e.g., `repaint(picture)`, the picture will update.

One of the important things that you can do with colors is to compare them. Some recipes for manipulating pictures will do *different* things with pixels depending on the color of the pixel. There are several ways of comparing pictures.

One way of comparing colors is the same way that one would compare numbers. We can subtract one color from the other. If we do that, we get a new color whose red, green, and blue components are the differences of each. So, if $color_1$ has red, green, and blue components (r_1, g_1, b_1), and $color_2$ has (r_2, g_2, b_2), then $color_1 - color_2$ creates a new color $(r_1 - r_2, g_1 - g_2, b_1 - b_2)$. We can also use `<`, `>`, and `==` (test for equality) to compare colors.

```
>>> print c1
color r=10 g=10 b=10
>>> print c2
color r=20 g=20 b=20
>>> print c2-c1
color r=10 g=10 b=10
>>> print c2 > c1
1
>>> print c2 < c1
0
```

Another method of comparing pictures is with a notion of color `distance`. You often won't care about an *exact* match of colors—two shades of blue might be *close enough* for your purposes. `distance` lets you measure close enough.

```
>>> print color
color r=81 g=63 b=51
>>> print newcolor
color r=255 g=51 b=51
>>> print distance(color,newcolor)
174.41330224498358
```

The distance between two colors is the Cartesian distance between the colors as points in a three-dimensional space, where red, green, and blue are the three dimensions. Recall that the distance between two points (x_1, y_1) and (x_2, y_2) is:

$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

The similar measure for two colors $(red_1, green_1, blue_1)$ and $(red_2, green_2, blue_2)$ is:

$\sqrt{(red_1 - red_2)^2 + (green_1 - green_2)^2 + (blue_1 - blue_2)^2}$

You can automatically get the darker or lighter versions of colors with `makeDarker` or `makeLighter`. (Remember that this was easy in HSV, but not so easy in RGB. These functions do it for you.)

```
>>> print color
color r=168 g=131 b=105
>>> print makeDarker(color)
color r=117 g=91 b=73
>>> print color
color r=117 g=91 b=73
```

You can also make colors from `pickAColor`, which gives you a variety of ways of picking a color.

```
>>> newcolor=pickAColor()
>>> print newcolor
color r=255 g=51 b=51
```

Once you have a color, you can get lighter or darker versions of the same color with `makeLighter` and `makeDarker`.

```
>>> print c
color r=10 g=100 b=200
>>> print makeLighter(c)
color r=10 g=100 b=200
>>> print c
color r=14 g=142 b=255
>>> print makeDarker(c)
color r=9 g=99 b=178
>>> print c
color r=9 g=99 b=178
```

When you have finished manipulating a picture, you can write it out with `writePictureTo`.

```
>>> writePictureTo(picture,"/Users/guzdial/newpicture.jpg")
```

Common Bug: End with .jpg
Be sure to end your filename with ".jpg" in order to get your operating system to recognize it as a JPEG file.

Of course, we don't have to write new functions to manipulate pictures. We can do it from the command area using the functions just described.

```
>>> file="/Users/guzdial/mediasources/barbara.jpg"
>>> pict=makePicture(file)
>>> show(pict)
>>> setColor(getPixel(pict,10,100),yellow)
>>> setColor(getPixel(pict,11,100),yellow)
>>> setColor(getPixel(pict,12,100),yellow)
>>> setColor(getPixel(pict,13,100),yellow)
>>> repaint(pict)
```

The result showing a small yellow line on the left side appears in Figure 5.7. This is 100 pixels down, and the pixels 10, 11, 12, and 13 from the left edge.

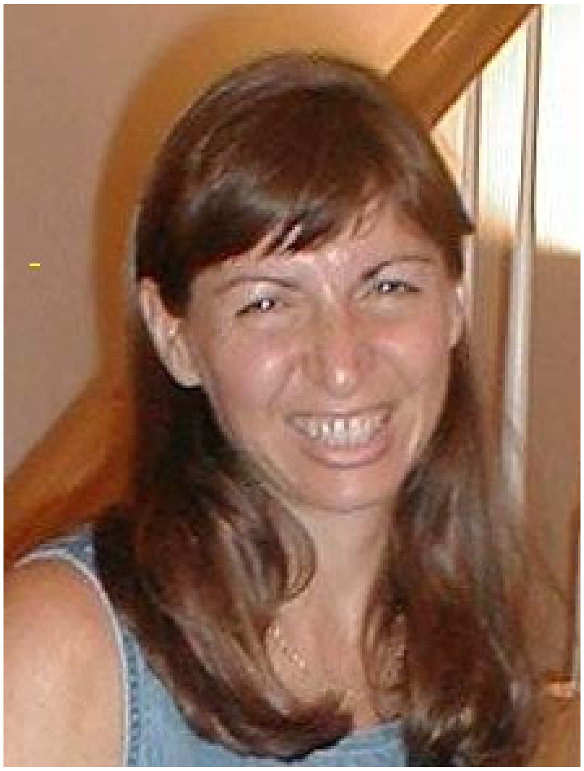


Figure 5.7: Directly modifying the pixel colors via commands: Note the small yellow line on the left

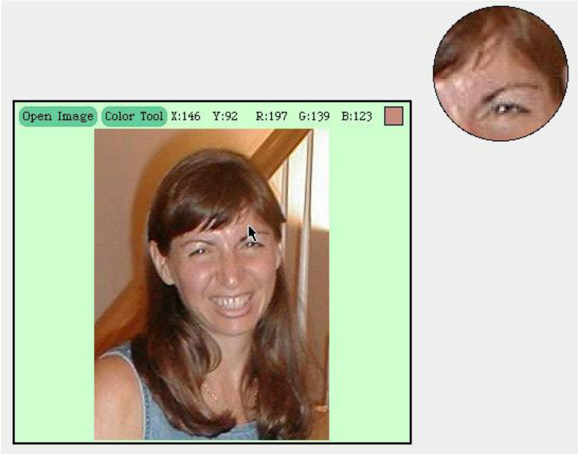
5.2.1 Exploring pictures

The MediaTools has a set of image exploration tools that are really useful for studying a picture (Figure 5.8). Use the Open button to bring up a file selection box, like you did for sounds. When the image appears, you have several different tools available. Move your cursor over the picture and press down with the mouse button.

- The red, green, and blue values will be displayed for the pixel you're pointing at. This is useful when you want to get a sense of how the colors in your picture map to numeric red, green, and blue values. It's also helpful if you're going to be doing some computation on the pixels and want to check the values.

- The x and y position will be display for the pixel you're point at. This is useful when you want to figure out regions of the screen, e.g., if you want to process only part of the picture. If you know the range of x and y coordinates where you want to process, you can tune your `for` loop to reach just those sections.

- Finally, a magnifier is available to let you see the pixels blown up. (The magnifier can be clicked and dragged around.)



X:146 Y:92 R:197 G:139 B:123

Figure 5.8: Using the MediaTools image exploration tools

5.2.2 Changing color values

The easiest thing to do with pictures is to change the color values of their pixels by changing the red, green, and blue components. You can get radically different effects by simply tweaking those values. Many of Adobe Photoshop's *filters* do just what we're going to be doing in this section.

Increasing/decreasing red (green, blue)

A common desire when working with digital pictures is to shift the *redness* (or greenness or blueness—but most often, redness) of a picture. You might shift it higher to "warm" the picture, or to reduce it to "cool" the picture or deal with overly-red digital cameras.

The below recipe reduces the amount of color 50% in an input picture.

Recipe 25: Reduce the amount of red in a picture by 50%

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

The recipe was used like this:

```
>>> file="/Users/guzdial/mediasources/barbara.jpg"

>>> picture=makePicture(file)

>>> show(picture)

>>> decreaseRed(picture)

>>> repaint(picture)
```

Common Bug: Patience: `for` loops always end
The most common bug with this kind of code is to give up and hit the Stop button before it stops. If you're using a `for` loop, the program will *always* stop. But it might take a full minute (or two!) for some of the manipulations we'll do—especially if your source image is large.

The original picture and its red-reduced version appear in Figure 5.9 (and at Figure 5.28 on page 157). 50% is obviously a *lot* of red to reduce! The picture looks like it was taken through a blue filter.

Let's increase the red in the picture now. If multiplying the red component by 0.5 reduced it, multiplying it by something over 1.0 should increase it. I'm going to apply the increase to the exact same picture, to see if we can even it out some (Figure 5.10 and Figure 5.29).

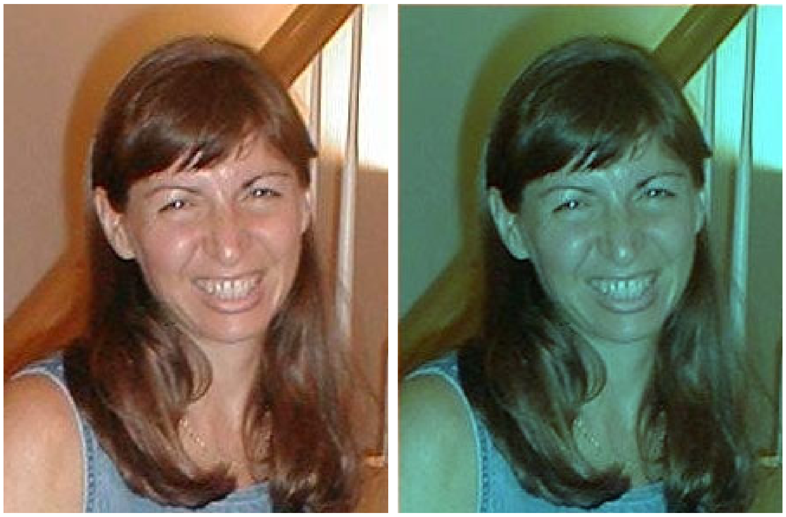


Figure 5.9: The original picture (left) and red-reduced version (right)

Recipe 26: Increase the red component by 20%

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

We can even get rid of a color completely. The below recipe erases the blue component from a picture (Figure 5.11 and Figure 5.30).

Recipe 27: Clear the blue component from a picture

```
def clearBlue(picture):
  for p in getPixels(picture):
    setBlue(p,0)
```

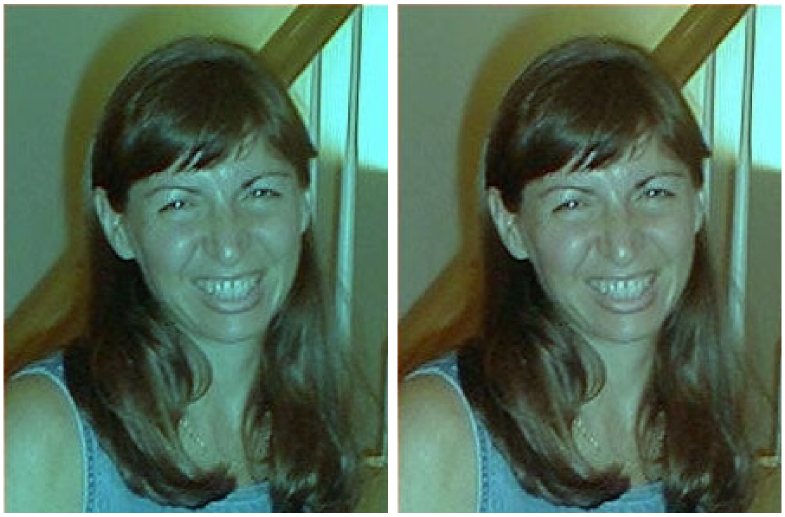


Figure 5.10: Overly blue (left) and red increased by 20% (right)

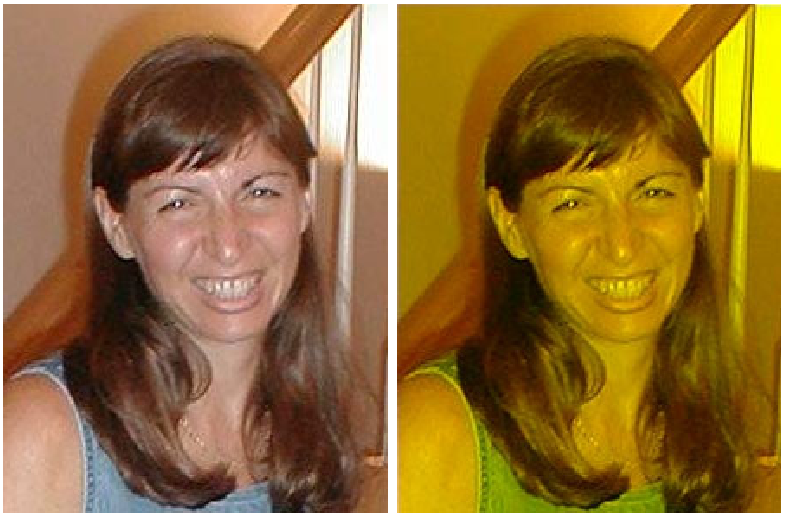


Figure 5.11: Original (left) and blue erased (right)

Lightening and darkening

To lighten or darken a picture is pretty simple. It's the same pattern as we saw previously, but instead of changing a color component, you change the overall color. Here's lightening and then darkening as recipes. Figure 5.12 (Figure 5.31) shows the lighter and darker versions of the original picture seen earlier.

Recipe 28: Lighten the picture

```
def lighten(picture):
  for px in getPixels(picture):
    color = getColor(px)
    makeLighter(color)
    setColor(px,color)
```

Recipe 29: Darken the picture

```
def darken(picture):
  for px in getPixels(picture):
    color = getColor(px)
    makeDarker(color)
    setColor(px,color)
```

Creating a negative

Creating a *negative image* of a picture is much easier than you might think at first. Let's think it through. What we want is the opposite of each of the current values for red, green, and blue. It's easiest to understand at the extremes. If we have a red component of 0, we want 255 instead. If we have 255, we want the negative to have a zero.

Now let's consider the middle ground. If the red component is slightly red (say, 50), we want something that is almost completely red—where the "almost" is the same amount of redness in the original picture. We want the maximum red (255), but 50 less than that. We want a red component of $255 - 50 = 205$. In general, the negative should be $255 - original$. We need

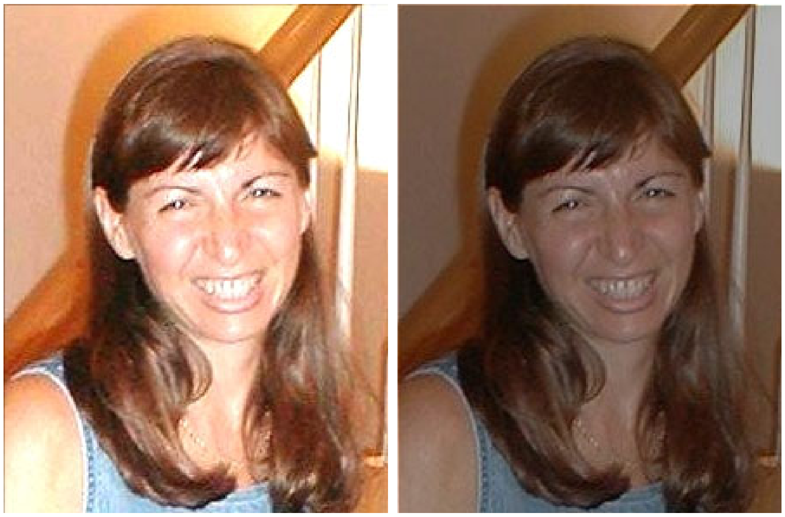


Figure 5.12: Lightening and darkening of original picture

to compute the negative of each of the red, green, and blue components, then create a new negative color, and set the pixel to the negative color.

Here's the recipe that does it, and you can see even from the grayscale image that it really does work (Figure 5.13 and Figure 5.32).

Recipe 30: Create the negative of the original picture

```
def negative(picture):
  for px in getPixels(picture):
    red=getRed(px)
    green=getGreen(px)
    blue=getBlue(px)
    negColor=makeColor( 255-red, 255-green, 255-blue)
    setColor(px,negColor)
```

Converting to greyscale

Converting to greyscale is a fun recipe. It's short, not hard to understand, and yet has such a nice visual effect. It's a really nice example of what one

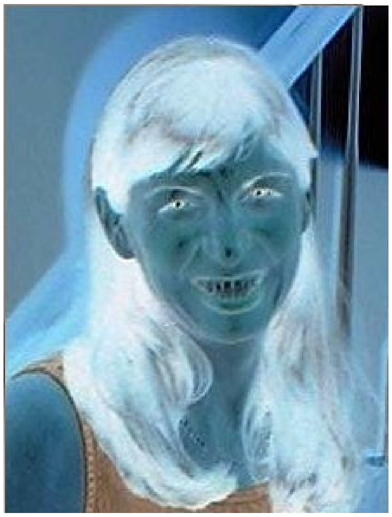


Figure 5.13: Negative of the image

can do easily yet powerfully by manipulating pixel color values.

Recall that the resultant color is grey whenever the red component, green component, and blue component have the same value. That means that our RGB encoding supports 256 levels of grey from, $(0,0,0)$ (black) to $(1,1,1)$ through $(100,100,100)$ and finally $(255,255,255)$. The tricky part is figuring out what the replicated value should be.

What we want is a sense of the *intensity* of the color. It turns out that it's pretty easy to compute: We average the three component colors. Since there are three components, the formula for intensity is:

$\frac{(red+green+blue)}{3}$

This leads us to the following simple recipe and Figure 5.14 (and Figure 5.33 on page 159).

Recipe 31: Convert to greyscale

```
def greyScale(picture):
  for p in getPixels(picture):
    intensity = (getRed(p)+getGreen(p)+getBlue(p))/3
    setColor(p,makeColor(intensity,intensity,intensity))
```

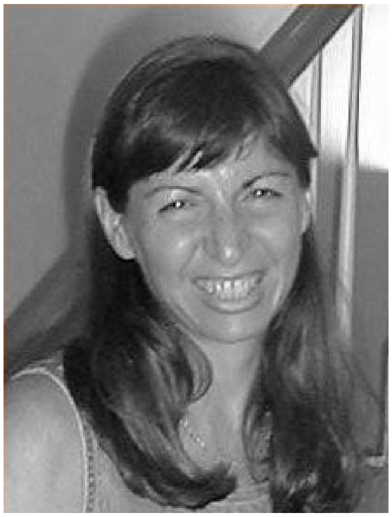


Figure 5.14: Color picture converted to greyscale

This is an overly simply notion of greyscale. Below is a recipe that takes into account how the human eye percieves *luminance*.

Recipe 32: Convert to greyscale with more careful control of luminance

```
def greyScaleNew(picture):
  for px in getPixels(picture):
    newRed = getRed(px) * 0.299
    newGreen = getGreen(px) * 0.587
    newBlue = getBlue(px) * 0.114
    luminance = newRed+newGreen+newBlue
    setColor(px,makeColor(luminance,luminance,luminance))
```

5.2.3 Copying pixels

With sounds, we could only get so far with `getSamples` before we had to start using more sophisticated `for` loops. Similarly, we can only get so far in our image processing with `getPixels`, so we'll need to start building our own for loops using range. Once we start doing that, we have more control over the exact *x* and *y* coordinates that we're processing, so we can start moving pixels where we want them, which is very powerful.

Looping across the pixels with range

Unlike sounds and samples, we can't use just a single for loop if we want to address every pixel. We have to use *two* of for loops—one to move horizontally across the pixels, and the other to move vertically to get every pixel. The function getPixels did this inside itself, to make it easier to write simple picture manipulations. But if you want to access each individual pixel, you'll need to use two loops, one for each dimension of the picture. The inner loop will be nested inside the outer loop, literally, inside its block. At this point, you're going to have to be careful in how you space your code to make sure that your blocks line up right.

Your loops will look something like this:

```
for x in range(1,getWidth(picture)):
  for y in range(1,getHeight(picture)):
    pixel=getPixel(picture,x,y)
```

For example, here's Recipe 28 (page 128), but using explicit pixel references.

Recipe 33: Lighten the picture using nested loops

```
def lighten(picture):
  for x in range(1,getWidth(picture)):
    for y in range(1,getHeight(picture)):
      px = getPixel(picture,x,y)
      color = getColor(px)
      makeLighter(color)
      setColor(px,color)
```

Mirroring a picture

Let's start out with an interesting effect that isn't particularly useful, but it is fun. Let's mirror a picture along its vertical axis. In other words, imagine that you have a mirror, and you place it on a picture so that the left side of the picture shows up in the mirror. That's the effect that we're going to implement. We'll do it in a couple of different ways.

First, let's think through what we're going to do. We'll pick a horizontal `mirrorpoint`—halfway across the picture, `getWidth(picture)/2`. (We want this to be an integer, a whole number, so we'll apply `int` to it.) We'll have the *x* cooridinate move from 1 to the `mirrorpoint`. At each value of *x*, we want to copy the color at the pixel *x* pixels to the *left* of the `mirrorpoint` to the pixel *x* pixels to the *right* of the mirrorpoint. The left would be `mirrorpoint-x` and the right would be `mirrorpoint+x`. Take a look at Figure 5.15 to convince yourself that we'll actually reach every pixel using this scheme. Here's the actual recipe.

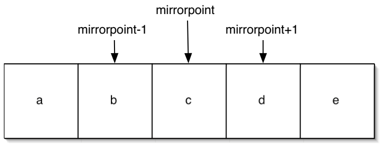


Figure 5.15: Once we pick a mirrorpoint, we can just walk *x* halfway and subtract/add to mirrorpoint

Recipe 34: Mirror pixels in a picture along a vertical line

```
def mirrorVertical(source):
  mirrorpoint = int(getWidth(source)/2)
  for y in range(1,getHeight(source)):
    for x in range(1,mirrorpoint):
      p = getPixel(source, x+mirrorpoint,y)
      p2 = getPixel(source, mirrorpoint-x,y)
      setColor(p,makeColor(getRed(p2), getGreen(p2),
getBlue(p2)))
```

We'd use it like this, and the result appears in Figure 5.16.

```
>>> file="/Users/guzdial/mediasources/santa.jpg"
>>> print file
/Users/guzdial/mediasources/santa.jpg
>>> picture=makePicture(file)
>>> mirrorVertical(picture)
>>> show(picture)
```

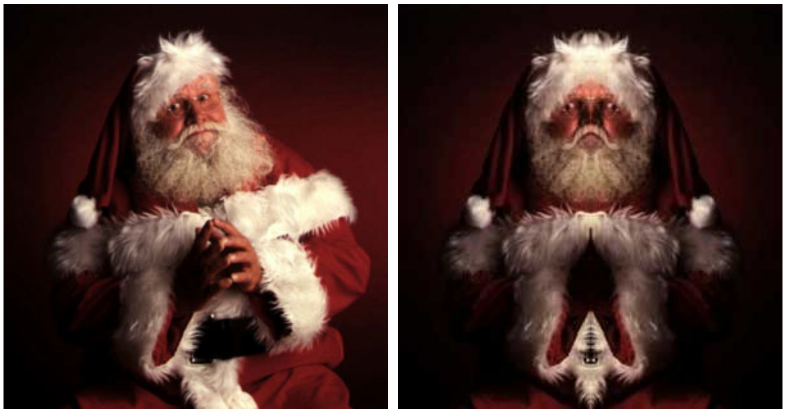


Figure 5.16: Original picture (left) and mirrored along the vertical axis (right)

We can do the same thing without the `mirrorpoint` by simply computing it as we move along. Here's the same recipe, shorter but more complex.

Recipe 35: Mirroring along the vertical axis, shorter

```
def mirrorVertical(source):
  for y in range(1, getHeight(source)):
    for x in range((getWidth(source)/2)+1,
getWidth(source)):
      p = getPixel(source,x,y)
      p2 = getPixel(source,(( getWidth(source)/2)- (x-(
getWidth(source)/2))),y)
      setColor(p,makeColor( getRed(p2), getGreen(p2),
getBlue(p2)))
```

Scaling a picture

A very common thing to do with pictures is to scale them. Scaling up means to make them larger, and scaling them down makes them smaller.

It's common to scale a 1-megapixel or 3-megapixel picture down to a smaller size to make it easier to place on the Web. Smaller pictures require less disk space, and thus less network bandwidth, and thus are easier and faster to download.

Scaling a picture requires the use of the *sampling sub-recipe* that we saw earlier. But instead of taking double-samples (to halve the frequency) or every-other-sample (to double the frequency), we'll be taking double-pixels (to double the size, that is, scale up) or every-other-pixel (to shrink the picture, that is, scale down).

XXX NEED TO EXPLAIN THIS BETTER

Our target will be the paper-sized JPEG file in the `mediasources` directory, which is 7x9.5 inches, which will fit on a 9x11.5 inch lettersize piece of paper with one inch margins.

```
>>> paperfile=getMediaPath("7inx95in.jpg")
>>> paperpicture=makePicture(paperfile)
>>> print getWidth(paperpicture)
504
>>> print getHeight(paperpicture)
684
```

Here's how this recipe works:

- We take as input a picture and an increment—a number of pixels to skip each time that we increment the source indices (x and y). If we use an increment less than 1.0, then the target picture grows larger. That's because we'll include the same pixel more than once. If we use an increment greater than 1.0, the target picture will be smaller.

- We set the `sourceX` index to start at one, and then start the `targetX` `for` loop.

- Same for the Y.

- We then copy the pixel color from the source to the target.

- *While inside the `targetY` loop*, we increment `sourceY`, and if it goes beyond the end of the picture, set it to the same amount as it went *beyond* the picture to the *beginng* of the picture.

- *Then while inside the `targetX` loop*, we do the same to `sourceX`.

Recipe 36: Scaling a picture

```
def resize(source,increment):
  newWidth = int(getWidth(source)* (1/increment))
  newHeight = int(getWidth(source)* (1/increment))
  target = makePicture( getMediaPath("7inx95in.jpg"))
  sourceX = 1
  for targetX in range(1,newWidth):
    sourceY = 1
    for targetY in range(1,newHeight):
      targetPixel=getPixel(target, targetX, targetY)
      sourcePixel=getPixel(source, int(sourceX),
int(sourceY))
      setColor(targetPixel,getColor(sourcePixel))
      sourceY = sourceY + increment
      if sourceY > getHeight(source):
        sourceY = sourceY- getHeight(source)
    sourceX = sourceX + increment
    if sourceX > getWidth(source):
      sourceX = sourceX- getWidth(source)

  return target
```

Here's a shorter version that does the same thing.

Recipe 37: Scaling, shorter

```
#Resizes a picture by a given positve real > 0
#such that 1 is the image unalterd, 2 is the image double
in size,
#0.5 is the image sized down in half
#so far only both the width and height can be altered at
once(not one seperate of the other)
#basic formula is old(x or y) / szMult = new(x or y)
#this code takes the new picture and for every pixel get a
coresponding pixel from the original
#szMult > 1 reuses pixels, while szMult < 1 skips pixels
#this is the simplest way to go about resizing an image,
the result will look blocky
#better algorithims involve linear interpolation and other
such ways to smoth the image and
#fill in the gaps, one thing that does help is to blur the
enlarged images to smooth them.
def resize(szMult):
pic = makePicture(pickAFile())
print(’old szw ’ , getWidth(pic) , ’ old szH ’
,getHeight(pic))
szW = getWidth(pic) * szMult
szH = getHeight(pic) * szMult
szW = int(szW)
szH = int(szH)
print(’new szw ’ , szW , ’ new szH ’ , szH)
newPic = BlankPicture(szW,szH)
for x in range(0,getWidth(newPic)):
for y in range(0,getHeight(newPic)):
setColor(getPixel(newPic,x,y), getColor( getPixel( pic,
int(x/szMult), int(y/szMult))))
return newPic
```

Creating a collage

In the `mediasources` folder are a couple images of flowers (Figure 5.17), each 100 pixels wide. Let's make a *collage* of them, by combining several of our effects to create different flowers. We'll copy them all into the blank image `640x480.jpg`. All we really have to do is to copy the pixel colors to the right places. The actual recipe is very long, so we'll put it at the end of the section.

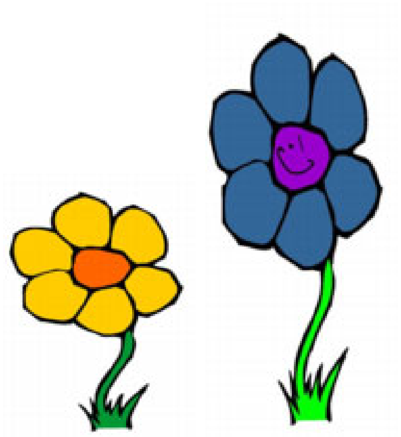


Figure 5.17: Flowers in the `mediasources` folder

Here's how we run the collage(Figure 5.18):

```
>>> flowers=createCollage()
Picture, filename /Users/guzdial/mediasources/flower1.jpg height
138 width 100
Picture, filename /Users/guzdial/mediasources/flower2.jpg height
227 width 100
Picture, filename /Users/guzdial/mediasources/640x480.jpg height
480 width 640
```

As long as this is, it would be even longer if we actually put all the effects in the same function. Instead, I copied the functions we did earlier. My whole program area looks like this:

```
def createCollage():
  flower1=makePicture(getMediaPath("flower1.jpg"))
  print flower1
```

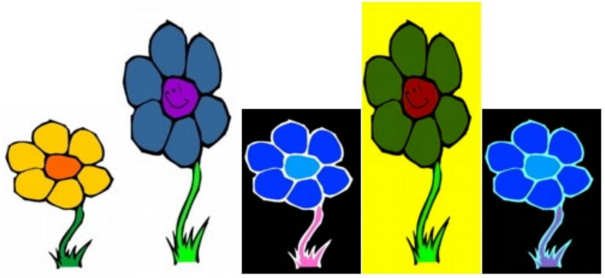


Figure 5.18: Collage of flowers

```
flower2=makePicture(getMediaPath("flower2.jpg"))
print flower2
canvas=makePicture(getMediaPath("640x480.jpg"))
print canvas
#First picture, at left edge
targetX=1
for sourceX in range(1,getWidth(flower1)):
  targetY=getHeight(canvas)-getHeight(flower1)-5
  for sourceY in range(1,getHeight(flower1)):
    px=getPixel(flower1,sourceX,sourceY)
    cx=getPixel(canvas,targetX,targetY)
    setColor(cx,getColor(px))
    targetY=targetY + 1
  targetX=targetX + 1
#Second picture, 100 pixels over
targetX=100
for sourceX in range(1,getWidth(flower2)):
  targetY=getHeight(canvas)-getHeight(flower2)-5
  for sourceY in range(1,getHeight(flower2)):
    px=getPixel(flower2,sourceX,sourceY)
    cx=getPixel(canvas,targetX,targetY)
    setColor(cx,getColor(px))
    targetY=targetY + 1
  targetX=targetX + 1
#Third picture, flower1 negated
negative(flower1)
targetX=200
for sourceX in range(1,getWidth(flower1)):
  targetY=getHeight(canvas)-getHeight(flower1)-5
  for sourceY in range(1,getHeight(flower1)):
    px=getPixel(flower1,sourceX,sourceY)
    cx=getPixel(canvas,targetX,targetY)
    setColor(cx,getColor(px))
    targetY=targetY + 1
  targetX=targetX + 1
#Fourth picture, flower2 with no blue
clearBlue(flower2)
targetX=300
for sourceX in range(1,getWidth(flower2)):
  targetY=getHeight(canvas)-getHeight(flower2)-5
  for sourceY in range(1,getHeight(flower2)):
    px=getPixel(flower2,sourceX,sourceY)
    cx=getPixel(canvas,targetX,targetY)
    setColor(cx,getColor(px))
    targetY=targetY + 1
  targetX=targetX + 1
#Fifth picture, flower1, negated with decreased red
decreaseRed(flower1)
targetX=400
for sourceX in range(1,getWidth(flower1)):
  targetY=getHeight(canvas)-getHeight(flower1)-5
  for sourceY in range(1,getHeight(flower1)):
    px=getPixel(flower1,sourceX,sourceY)
    cx=getPixel(canvas,targetX,targetY)
    setColor(cx,getColor(px))
    targetY=targetY + 1
  targetX=targetX + 1
show(canvas)
return(canvas)
```

```
def clearBlue(picture):
  for p in getPixels(picture):
    setBlue(p,0)

def negative(picture):
  for px in getPixels(picture):
    red=getRed(px)
    green=getGreen(px)
    blue=getBlue(px)
    negColor=makeColor( 255-red, 255-green, 255-blue)
    setColor(px,negColor)

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

Recipe 38: Creating a collage

```
def createCollage():
  flower1=makePicture(getMediaPath("flower1.jpg"))
  print flower1
  flower2=makePicture(getMediaPath("flower2.jpg"))
  print flower2
  canvas=makePicture(getMediaPath("640x480.jpg"))
  print canvas
  #First picture, at left edge
  targetX=1
  for sourceX in range(1,getWidth(flower1)):
    targetY=getHeight(canvas)-getHeight(flower1)-5
    for sourceY in range(1,getHeight(flower1)):
      px=getPixel(flower1,sourceX,sourceY)
      cx=getPixel(canvas,targetX,targetY)
      setColor(cx,getColor(px))
      targetY=targetY + 1
    targetX=targetX + 1
  #Second picture, 100 pixels over
  targetX=100
  for sourceX in range(1,getWidth(flower2)):
    targetY=getHeight(canvas)-getHeight(flower2)-5
    for sourceY in range(1,getHeight(flower2)):
      px=getPixel(flower2,sourceX,sourceY)
      cx=getPixel(canvas,targetX,targetY)
      setColor(cx,getColor(px))
      targetY=targetY + 1
    targetX=targetX + 1
  #Third picture, flower1 negated
  negative(flower1)
  targetX=200
  for sourceX in range(1,getWidth(flower1)):
    targetY=getHeight(canvas)-getHeight(flower1)-5
    for sourceY in range(1,getHeight(flower1)):
      px=getPixel(flower1,sourceX,sourceY)
      cx=getPixel(canvas,targetX,targetY)
      setColor(cx,getColor(px))
      targetY=targetY + 1
    targetX=targetX + 1
  #Fourth picture, flower2 with no blue
  clearBlue(flower2)
  targetX=300
  for sourceX in range(1,getWidth(flower2)):
    targetY=getHeight(canvas)-getHeight(flower2)-5
    for sourceY in range(1,getHeight(flower2)):
      px=getPixel(flower2,sourceX,sourceY)
      cx=getPixel(canvas,targetX,targetY)
      setColor(cx,getColor(px))
      targetY=targetY + 1
    targetX=targetX + 1
  #Fifth picture, flower1 negated with decreased red
```

5.2.4 Replacing Colors

Replacing colors with another color is pretty easy. We can do it broadly, or just within a range.

Here's a recipe that tries to replace the brown color in Barbara's hair with red. I used the MediaTools pixel tools to figure out roughly what the RGB values were for Barb's brown hair, then wrote a program to look for colors close to that, and increase the redness of those pixels. I played a lot with the value that I used for distance (here, 50.0) and the amount of redness increase (here, 50% increase). The result is that the wood behind her gets increased, too (Figure 5.19 and Figure 5.34).

Recipe 39: Color replacement: Turn Barbara into a redhead

```
def turnRed():
  brown = makeColor(57,16,8)
  file="/Users/guzdial/mediasources/barbara.jpg"
  picture=makePicture(file)
  for px in getPixels(picture):
    color = getColor(px)
    if distance(color,brown)<50.0:
      redness=int(getRed(px)*1.5)
      blueness=getBlue(px)
      greenness=getGreen(px)
      setColor(px,makeColor(redness,blueness,greenness))
  show(picture)
  return(picture)
```

With the MediaTools, we can also figure out the coordinates just around Barb's face, and then just do the browns near her face. The effect isn't too good, though it's clear that it worked. The line of redness is too sharp and rectangular (Figure 5.20 and Figure 5.35).

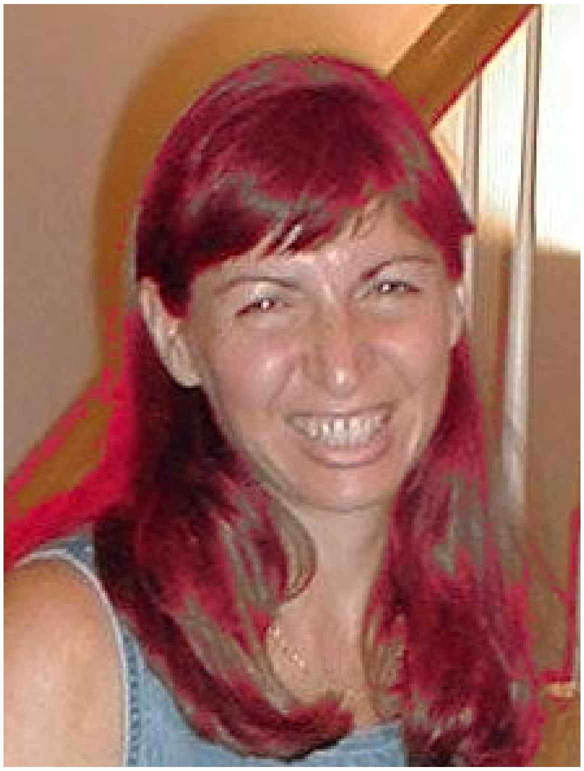


Figure 5.19: Increasing reds in the browns

Recipe 40: Color replacement in a range

```
def turnRedInRange():
  brown = makeColor(57,16,8)
  file="/Users/guzdial/mediasources/barbara.jpg"
  picture=makePicture(file)
  for x in range(70,168):
    for y in range(56,190):
      px=getPixel(picture,x,y)
      color = getColor(px)
      if distance(color,brown)<50.0:
        redness=int(getRed(px)*1.5)
        blueness=getBlue(px)
        greenness=getGreen(px)
        setColor(px,makeColor(redness,blueness,greenness))
  show(picture)
  return(picture)
```

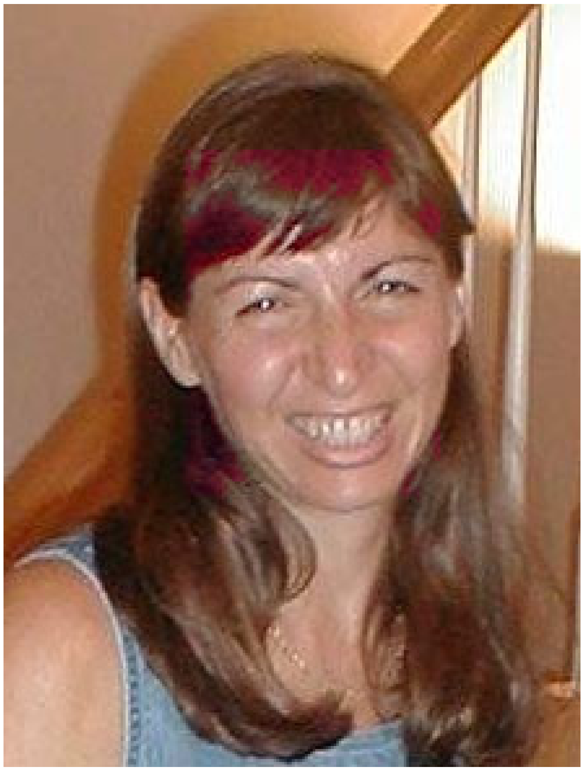


Figure 5.20: Increasing reds in the browns, within a certain range

Background subtraction

Let's imagine that you have a picture of someone, and a picture of where they stood without them there (Figure 5.21). Could you *subtract* the background of the person (i.e., figure out where the colors are exactly the same), and then replace another background? Say, of the moon (Figure 5.22)?

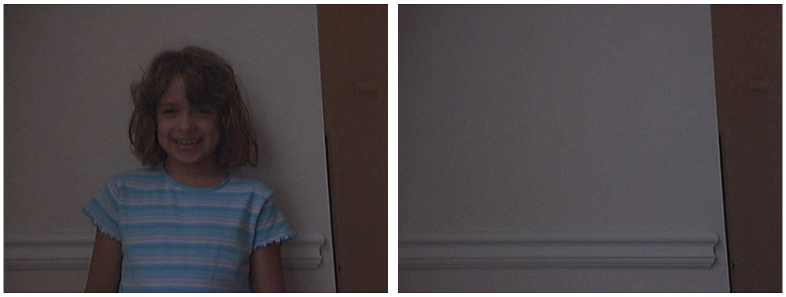


Figure 5.21: A picture of a child (Katie), and her background without her

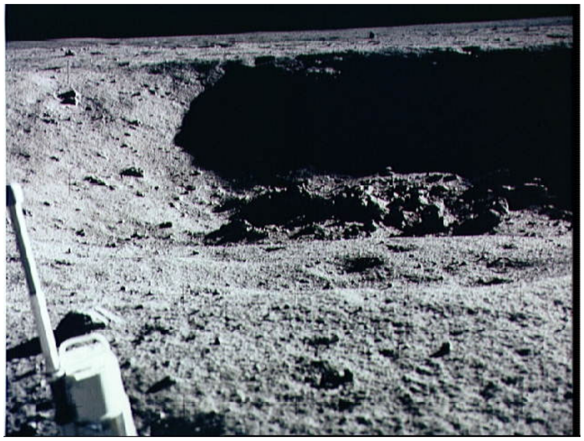


Figure 5.22: A new background, the moon

Recipe 41: Subtract the background and replace it with a new one

```
#Picture with person, background, and newbackground
def swapbg(pic1, bg, newbg):
  for x in range(1,getWidth(pic1)):
    for y in range(1,getHeight(pic1)):
      p1px = getPixel(pic1,x,y)
      bgpx = getPixel(bg,x,y)
      if (distance(getColor(p1px),getColor(bgpx)) < 15.0):
        setColor(p1px,getColor(getPixel(newbg,x,y)))
  return pic1
```

You can, but the effect isn't as good as you'd like (Figure 5.23). My daughter's shirt color was too close to the color of the wall. And though the light was dim, a shadow is definitely having an effect here.

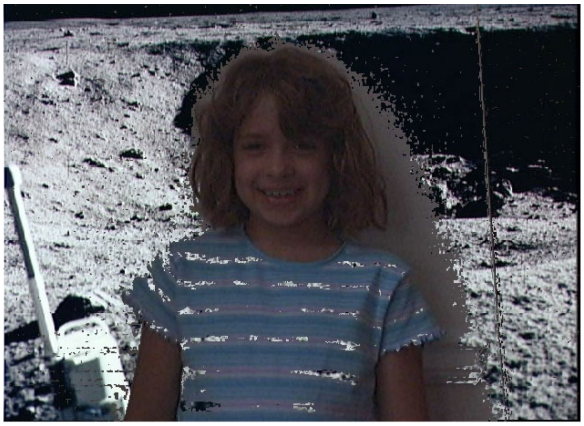


Figure 5.23: Katie on the moon

Chromakey

The way that weatherpersons do it is to stand before a background of a fixed color (usually blue or green), then subtract that color. This is called *chromakey*. I took my son's blue sheet, attached it to the entertainment center, then took a picture of myself in front of it using a timer on a camera (Figure 5.24).

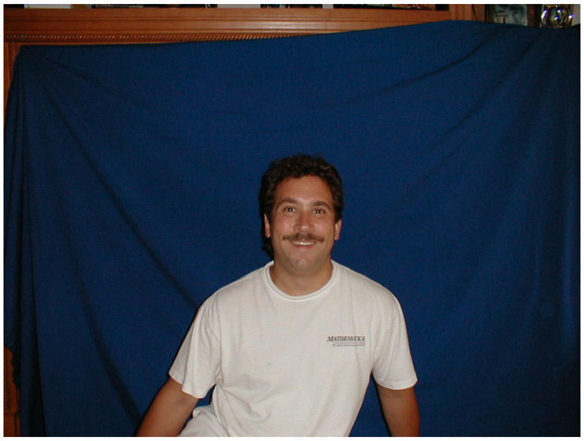


Figure 5.24: Mark in front of a blue sheet

Recipe 42: Chromakey: Replace all blue with the new background

```
def chromakey(source,bg):
 # source should have something in front of blue, bg is
the new background
 for x in range(1,getWidth(source)):
  for y in range(1,getHeight(source)):
   p = getPixel(source,x,y)
   # My definition of blue: If the redness + greenness <
blueness
   if (getRed(p) + getGreen(p) < getBlue(p)):
   #Then, grab the color at the same spot from the new
background
    setColor(p,getColor(getPixel(bg,x,y)))
 return source
```

The effect is really quite striking (Figure 5.25). Do note the "folds" in the lunar surface, though. The really cool thing is that this recipe works for any background that's the same size as the image (Figure 5.26).

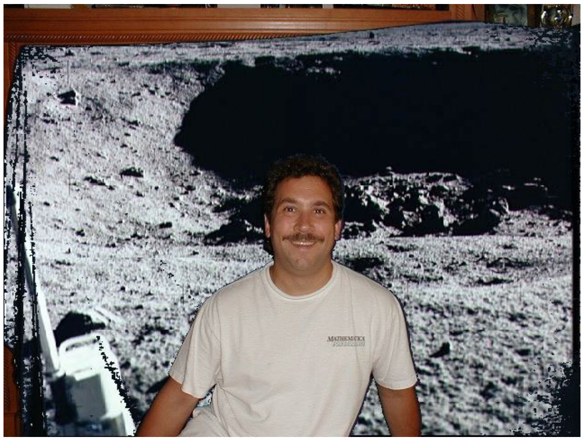


Figure 5.25: Mark on the moon

There's another way of writing this code, which is shorter but does the same thing.

Recipe 43: Chromakey, shorter

```
def chromakey2(source,bg):
  for p in pixels(source):
    if (getRed(p)+getGreen(p) < getBlue(p)):
      setColor(p,getColor(getPixel(bg,x(p),y(p))))
  return source
```

5.2.5 Combining pixels

Some of our best effects come from combining pixels—using pixels from other sides of a target pixel to inform what we do.

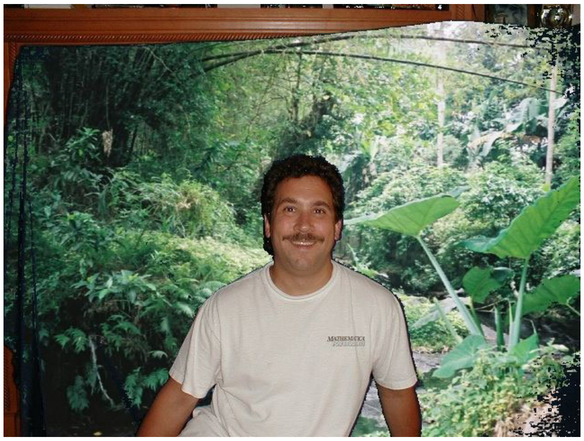


Figure 5.26: Mark in the jungle

Blurring

Recipe 44: Blurring a picture to get rid of rough edges

```
def blurExample(size):
  pic = makePicture(pickAFile())
  newPic = blur(pic,size)
  show(newPic)
  show(pic)

#!!
#To blur an image we take a pixel and set it’s color to the
average of all pixels around
#it(of a certain distance) to the current pixels color.
This give the effect of
#blending things together such as in the case of a blur
where you lose detail.
#
#pic is the image, and size is how big an area to average,
1=3x3 pixel area with current pixel
#in the center, 2=5x5, 3=7x7...
#positive #s only , 0 will do nothing and return the
original image
def blur(pic,size):
  new = getFolderPath("640x480.jpg")
  for x in range(1,getWidth(pic)):
    print ’On x> ’, x
    for y in range(1,getHeight(pic)):
      newClr = blurHelper(pic,size,x-size,y-size)
      setColor(getPixel(new,x,y),newClr)
  return new
#!!
#At a given x,y(integer that is in the image) in the
picture,pic, is sums up the area
#of pixels as indicated by size.
#
```

Functions and Objects Summary

In this chapter, we talk about several kinds of encodings of data (or objects).

Pictures	Pictures are encodings of images, typically coming from a JPEG file.
Pixels	Pixels are a sequence of Pixel objects. They *flatten* the two dimensional nature of the pixels in a picture and give you instead an array-like sequence of pixels. `pixels[1]` returns the leftmost pixel in a picture.
Pixel	A pixel is a dot in the Picture. It has a color and an (x, y) position associated with it. It remembers its own Picture so that a change to the pixel changes the real dot in the picture.
Color	It's a mixture of red, green, and blue values, each between 0 and 255.

Here are the functions used or introduced in this chapter:

pickAFile	Lets the user pick a file and returns the complete path name as a string. No input
makePicture	Takes a filename as input, reads the file, and creates a picture from it. Returns the picture.
show	Shows a picture provided as input. No return value.
getPixels	Takes a picture as input and returns the sequence of Pixel objects in the picture.
getPixel	Takes a picture, an x position and a y position (two numbers), and returns the Pixel object at that point in the picture.
getWidth	Takes a picture as input and returns its width in the number of pixels across the picture.
getHeight	Takes a picture as input and returns its length in the number of pixels top-to-bottom in the picture.
writePictureTo	Takes a picture and a file name (string) as input, then writes the picture to the file as a JPEG. (Be sure to end the filename in ".jpg" for the operating system to understand it well.)
addText	Takes a picture, an x position and a y position (two numbers), and some text as a string, which will get drawn into the picture.
addLine	Takes a picture, a starting (x, y) position (two numbers), and an ending (x, y) position (two more numbers, four total) and draws a black line from the starting point to the ending point in the picture.
addRect	Takes a picture, a starting (x, y) position (two numbers), and a width and height (two more numbers, four total) then draws a black rectangle *in outline* of the given width and height with the position (x, y) as the upper left corner.
addRectFilled	Exactly like `addRect`, but fills the rectangle with black.

getRed, getGreen, getBlue	Each of these functions takes a Pixel object and returns the value (between 0 and 255) of the amount of redness, greenness, and blueness (respectively) in that pixel.
setRed, setGreen, setBlue	Each of these functions takes a Pixel object and a value (between 0 and 244) and sets the redness, greenness, or blueness (respectively) of that pixel to the given value.
getColor	Takes a Pixel and returns the Color object at that pixel.
setColor	Takes a Pixel object and a Color object and sets the color for that pixel.
getX, getY	Takes a Pixel object and returns the x or y (respectively) position of where that Pixel is at in the picture.

makeColor	Takes three inputs: For the red, green, and blue components (in order), then returns a color object.
pickAColor	Takes no input, but puts up a color picker. Find the color you want, and the function will return the Color object of what you picked.
distance	Takes two Color objects and returns a single number representing the distance between the colors. The red, green, and blue values of the colors are taken as a point in (x, y, z) space, and the cartesian distance is computed.
makeDarker, makeLighter	Each take a color and return a slightly darker or lighter (respectively) version of the color.

There are a bunch of *constants* that are useful in this chapter. These are variables with pre-defined values. These values are colors: `black, white, blue, red, green, gray, darkGray, lightGray, yellow, orange, pink, magenta, cyan`.

Exercises

Exercise 36: Recipe 25 (page 125) is obviously too much color reduction. Write a version that only reduces the red by 10%, then one by 20%. Which seems to be more useful? Note that you can always repeatedly reduce the redness in a picture, but you don't want to have to do it *too* many times, either.

Exercise 37: Write the blue and green versions of Recipe 25 (page 125).

Exercise 38: Each of the below is equivalent to Recipe 26 (page 126). Test them and convince them. Which do you prefer and why?

```
def increaseRed2(picture):
  for p in getPixels(picture):
    setRed(p,getRed(p)*1.2)

def increaseRed3(picture):
  for p in getPixels(picture):
    redComponent = getRed(p)
    greenComponent = getGreen(p)
    blueComponent = getBlue(p)
    newRed=int(redComponent*1.2)
    newColor = makeColor(newRed,greenComponent,blueComponent)
    setColor(p,newColor)
```

Exercise 39: If you keep increasing the red, eventually the red looks like it disappears, and you eventually get errors about illegal arguments. What you do think is going on?

Exercise 40: Rewrite Recipe 27 (page 126) to clear blue, but for red and green. For each of these, which would be the most useful in actual practice? How about combinations of these?

Exercise 41: Rewrite Recipe 27 (page 126) to *maximize* blue (i.e., setting it to 255) instead of clearing it. Is this useful? Would the red or green versions be useful?

Exercise 42: There is more than one way to compute the right greyscale value for a color value. The simple recipe that we use in Recipe 31 (page 130) may not be what your greyscale printer uses when printing a color picture. Compare the color (relatively unconverted by the printer) greyscale image using our simple algorithm in Figure 5.33 with the original color picture that the printer has converted to greyscale (left of Figure 5.9). How do the two pictures differ?

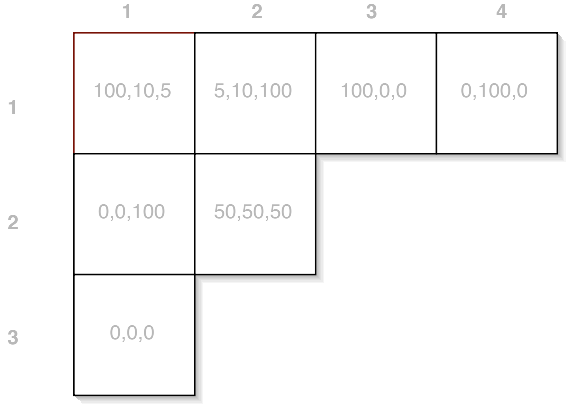
Exercise 43: Are Recipe 34 (page 133) and Recipe 35 (page 134) really the same? Look at them carefully and consider the *end conditions*: The points when x is at the beginning and end of its range, for example. It's easy in loops to be "off-by-one."

Exercise 44: Can you rewrite the vertical mirroring function (Recipe 34 (page 133)) to do horizontal mirroring? How about mirroring along the diagonal (from $(1, 1)$ to $(width, height)$?

Exercise 45: Think about how the greyscale algorithm works. Basically, if you know the *luminance* of anything visual (e.g., a small image, a letter), you can replace a pixel with that visual element in a similar way to create a collage image. Try implementing that. You'll need 256 visual elements of increasing lightness, all of the same size. You'll create a collage by replacing each pixel in the original image with one of these visual elements.

To Dig Deeper

5.3 Color Figures



	1	2	3	4
1	100,10,5	5,10,100	100,0,0	0,100,0
2	0,0,100	50,50,50		
3	0,0,0			

Figure 5.27: Color: RGB triplets in a matrix representation

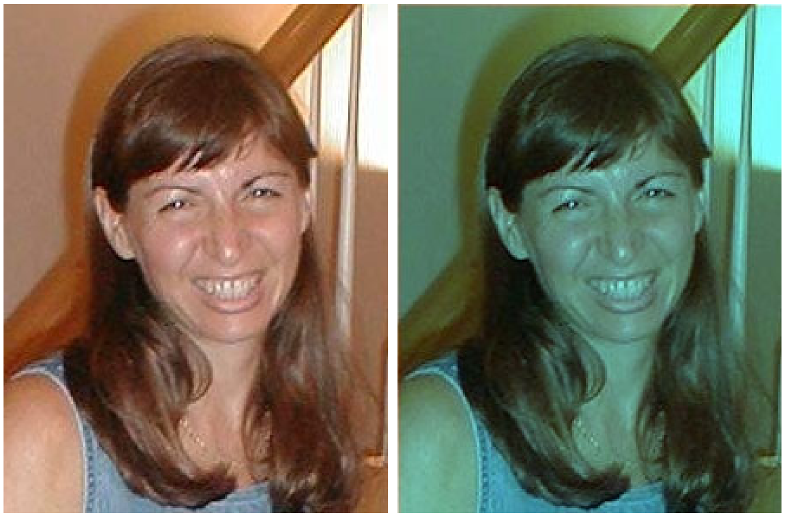


Figure 5.28: Color: The original picture (left) and red-reduced version (right)

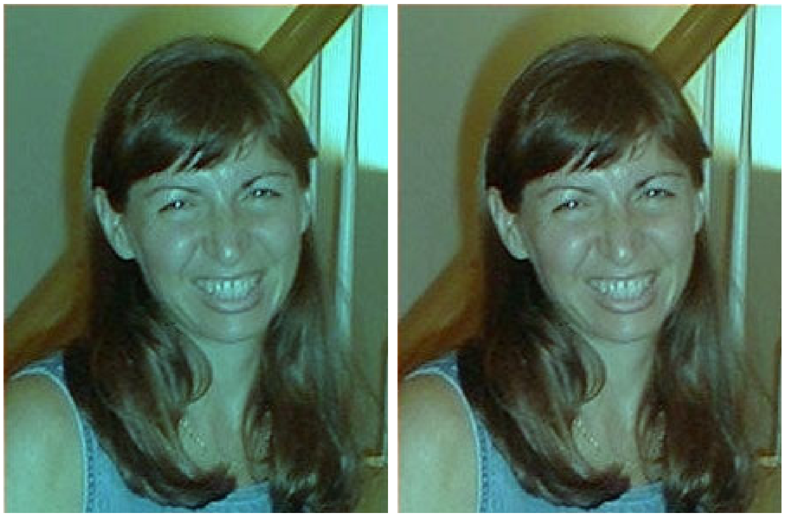


Figure 5.29: Color: Overly blue (left) and red increased by 20% (right)

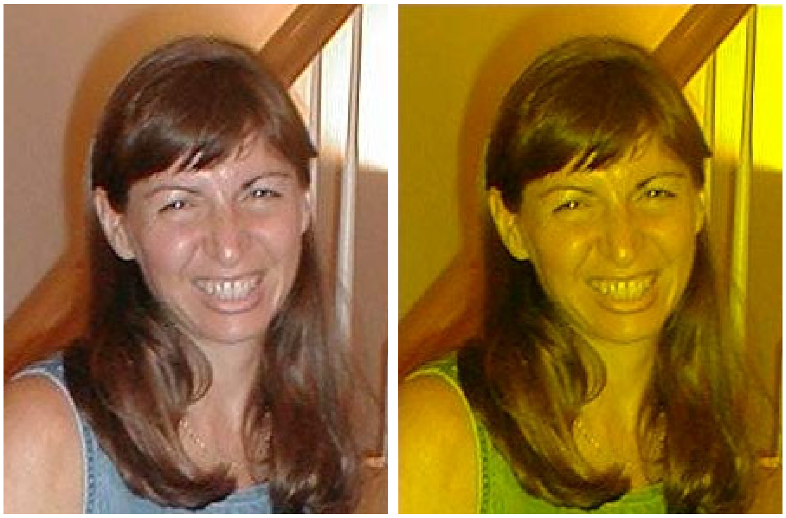


Figure 5.30: Color: Original (left) and blue erased (right)

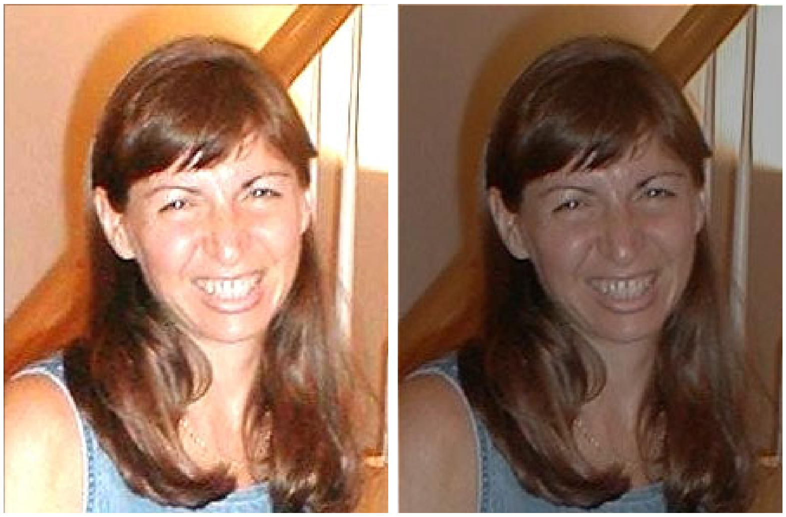


Figure 5.31: Color: Lightening and darkening of original picture

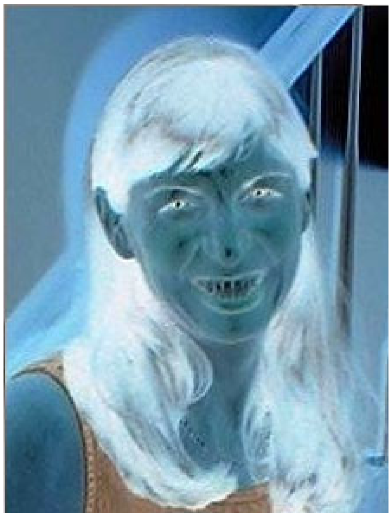


Figure 5.32: Color: Negative of the image

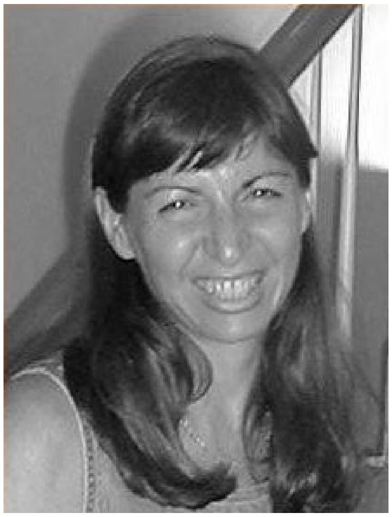


Figure 5.33: Color: Color picture converted to greyscale

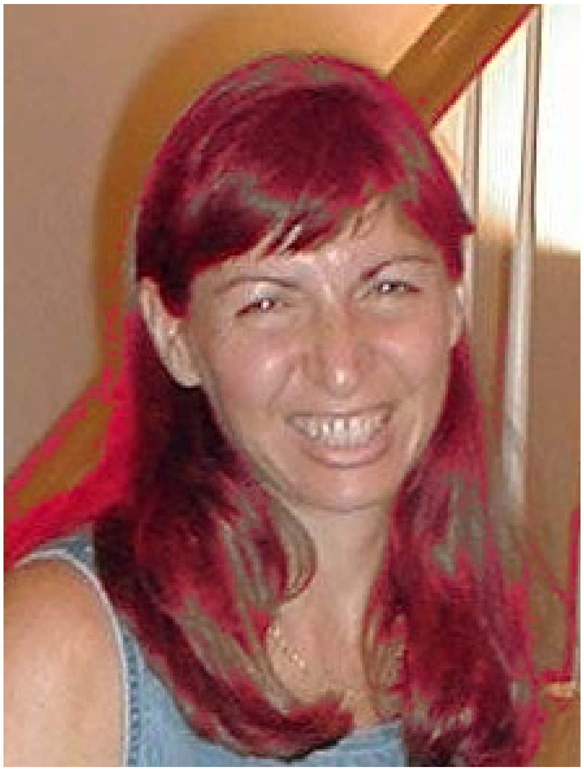


Figure 5.34: Color: Increasing reds in the browns

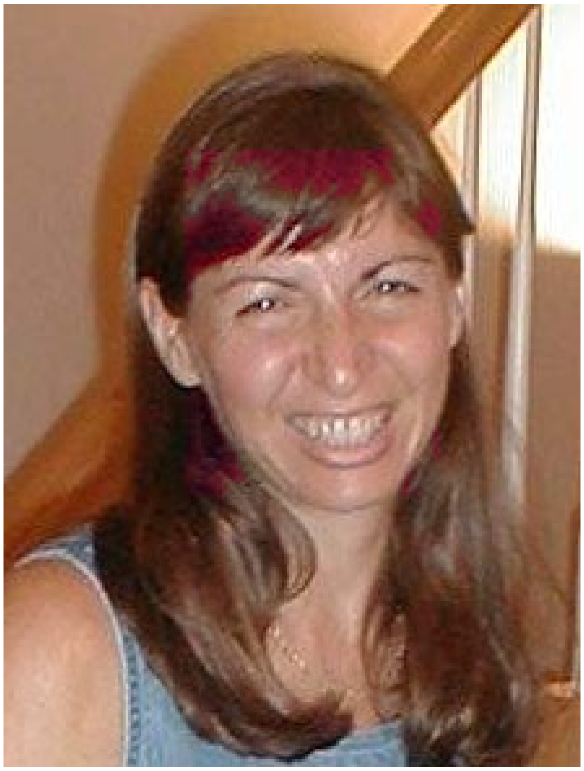


Figure 5.35: Color: Increasing reds in the browns, within a certain range

Chapter 6

Creating Pictures

Part IV

Meta-Issues: How we do what we do

Chapter 7

Design and Debugging

How do we do this? How do we make programs, and then make them actually *run*? This chapter is about some techniques for this.

7.1 Designing programs

7.1.1 Top-down

- Start out with the program statement. If you don't have one, write one. What are you trying to do?
- Start breaking it down. Are there parts to the program? Do you know that you have to open some pictures or set some constant values? And then are there some loops? How many do you need?
- Keep breaking it down until you get to statements or commands or functions that you know. Write those down.

7.1.2 Bottom-up

- What do you know how to do of your program? Does it say that you have to manipulate sound? Try a couple of the sound recipes in the book to remember how to do that. Does it say that you have to change red levels? Can you find a recipe that does that and try it?
- Now, can you add some of these together? Can you put together a couple of recipes that do *part* of what you want?
- Keep growing the program. Is it closer to what you need? What else do you need to add?

- Run your program *often*. Make sure it works, and that you understand what you have so far.
- Repeat until you're satisfied with the result.

7.2 Techniques of Debugging

How do you figure out what your program is doing, if it runs, but isn't doing what you want?

Tracing Code

Sit down with pencil and paper and figure out the variable values and what's happening.

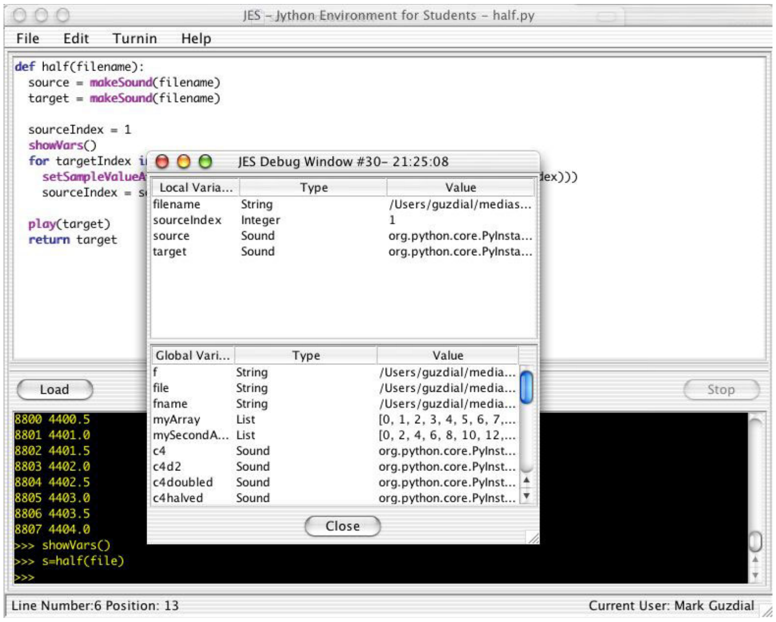
`Print` statements really are very useful to help one in tracing code.

Seeing the Variables

The `showVars` function will show you all the variables at that point in the program.

Exercises

To Dig Deeper


JES – Jython Environment for Students – half.py
File
Edit
Turnin
Help
JES Debug Window #30– 21:25:08
Load
Stop
Close

```
def half(filename):
  source = makeSound(filename)
  target = makeSound(filename)

  sourceIndex = 1
  showVars()
  for targetIndex i
    setSampleValueA                                        dex)))
    sourceIndex = s

  play(target)
  return target
```

Local Varia...	Type	Value
filename	String	/Users/guzdial/medias...
sourceIndex	Integer	1
source	Sound	org.python.core.PyInsta...
target	Sound	org.python.core.PyInsta...

Global Vari...	Type	Value
f	String	/Users/guzdial/media...
file	String	/Users/guzdial/media...
fname	String	/Users/guzdial/media...
myArray	List	[0, 1, 2, 3, 4, 5, 6, 7,...
mySecondA...	List	[0, 2, 4, 6, 8, 10, 12,...
c4	Sound	org.python.core.PyInst...
c4d2	Sound	org.python.core.PyInst...
c4doubled	Sound	org.python.core.PyInst...
c4halved	Sound	org.python.core.PyInst...

```
8800 4400.5
8801 4401.0
8802 4401.5
8803 4402.0
8804 4402.5
8805 4403.0
8806 4403.5
8807 4404.0
>>> showVars()
>>> s=half(file)
>>>
```

Line Number:6 Position: 13   Current User: Mark Guzdial

Figure 7.1: Seeing the variables using `showVars()`

Part V

Files

Chapter 8

Encoding, Creating, and Manipulating Files

This chapter will eventually talk about directories and manipulating directories, and how to read and write files.

8.1 How to walk through a directory

To manipulate a directory of files, you have to `import` a *module* that contains additional functionality. To manipulate the functions in the module, you'll have to use *dot notation*.

```
>>> print file
/Users/guzdial/Work/mediasources/dark-bladerunner/dark-bladerunner
001.jpg
>>> import os
>>> for file in os.listdir("/Users/guzdial/Work/mediasources/dark-bladerunner"):
...    print file
...
dark-bladerunner 001.jpg
dark-bladerunner 002.jpg
dark-bladerunner 003.jpg
...
```

Chapter 9

Moving Files

Part VI

Text

Manipulating text is what this chapter is about. It's very important for us because a very common form of text for us today is *HyperText Markup Language (HTML)*.

9.1 A recipe to generate HTML

Be sure to do `setMediaFolder()` before running this!

Recipe 45: A recipe to generate HTML

```
    def html():

        # To use this routine the Media folder must be used for
    your output and picture files

        myHTML =  getMediaPath("myHTML.html")
        pictureFile = getMediaPath("barbara.jpg")
        eol="\ n”
    # The following line builds a literal string that includes
both single and double quotes
    buildSpecial = "<IMG SRC=¨+ pictureFile + ’" ALT= "I am
one heck of a programmer!">’+eol

    outFile = open(myHTML,’w’)

    outFile.write( ’<HTML>’+eol)
    outFile.write(’<HEAD>’+eol)
    outFile.write(’<TITLE>Homepage of Georgia P.
Burdell</TITLE>’+eol)
    outFile.write(’<LINK REL=STYLESHEET TYPE="text/css"
HREF="style.css">’)
    outFile.write(’</HEAD>’+eol)
    outFile.write(’<BODY>’+eol)
    outFile.write(’<CENTER><H2>Welcome to the home page of
Georgia P. Burdell!</H2></CENTER>’+eol)
    outFile.write(’<BR>’+eol)
    outFile.write(’<P> Hello, and welcome to my home page! As
you should have already’+eol)
    outFile.write(’ guessed, my name is Georgia, and I am a <A
HREF=http://www.cc.gatech.edu><B>’+eol)
    outFile.write(’Computer Science</B></A> major at <A
HREF=http://www.gatech.edu><B>’+eol)
    outFile.write(’Georgia Tech</B> </A>’+eol)
    outFile.write(’<BR>’+eol)
    outFile.write(’Here is a picture of me in case you were
wondering what I looked like.’+eol)
    outFile.write(’</P>’+eol)
    outFile.write(buildSpecial)                                        #
Write the special line we built up near the top
    outFile.write(’<P><H4> Well, welcome to my web page. The
majority of it is still under construction, so I dont́ have a
lot to show you right now. ’+eol)
    outFile.write(’I am in my 75th year at Georgia Tech but
am taking CS 1315 so I dont́ have a lot of spare time to update
the page.’+eol)
    outFile.write(’I promise to start real soon!’+eol)
    outFile.write(’--Georgia P. Burdell</P></H4>’+eol)
    outFile.write(’<HR>’+eol)
    outFile.write(’<PIf you want to send me e-mail, click <><A
```

Chapter 10

Encoding and Manipulation of Text

Part VII

Movies

Chapter 11

Encoding, Manipulation and Creating Movies

Movies (video) are actually very simple to manipulate. They are arrays of pictures (*frames*). You need to be concerned with the *frame rate* (the number of frames per second), but it's mostly just things you've seen before.

It just takes a *long* time to process. . .

11.1 Manipulating movies

To manipulate movies, we have to break it into frames. The MediaTools can do that for you (Figure 11.1). The Menu button lets you save any *MPEG* movie as a series of JPEG frame pictures.

I've already broken up a section of the start of the movie *Bladerunner*

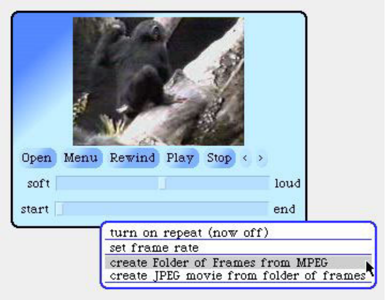


Figure 11.1: Movie tools in MediaTools

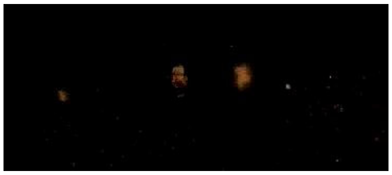


Figure 11.2: Dark starting frame number 9

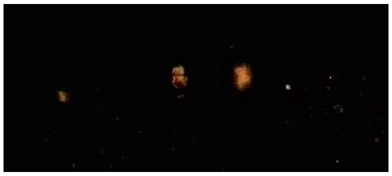


Figure 11.3: Somewhat lighter starting frame number 9

into a folder in the `mediasources` directory. You'll find that these are very dark frames (Figure 11.2). Can we lighten them (Figure 11.3)? Well, maybe a little.

Common Bug: If you see `getMediaPath`, then `setMediaFolder`
Whenever you see `getMediaPath` in a recipe, you know that you have to `setMediaFolder` before using that recipe.

You'd run this like `lightenMovie("/Users/guzdial/mediasources/dark-bladerunner/")`. *Be sure to include the final file directory delimeter!*.

Recipe 46: Lightening frames of a movie

```
def lightenMovie(folder):
  import os
  for file in os.listdir(folder):
    picture=makePicture(folder+file)
    for px in getPixels(picture):
      color=getColor(px)
      makeLighter(color)
      setColor(px,color)
    writePictureTo(picture,folder+"l"+file)
```

11.2 Compositing to create new movies

What if Santa were sneaking along just below the camera when *Bladerunner* was shot? (Or maybe it's a *Tribble*.)

Recipe 47: Compositing new images into movie frames

```
def santaMovie(folder):
  santafile="/Users/guzdial/Work/mediasources/santa.jpg"
  santa=makePicture(santafile)
  startXPos = 10
  startYPos = 100
  import os
  for file in os.listdir(folder):
    frame=makePicture(folder+file)
    xmax=min(startXPos+getWidth(santa),getWidth(frame))
    ymax=min(startYPos+getHeight(santa),getHeight(frame))
    santaX = 1
    for x in range(startXPos,xmax):
      santaY = 1
      for y in range(startYPos,ymax):
        px=getPixel(frame,x,y)
        santaPixel=getPixel(santa,santaX,santaY)
        setColor(px,getColor(santaPixel))
        santaY = santaY + 1
      santaX = santaX + 1
    writePictureTo(frame,folder+"s"+file)
    # make Santa sink one line lower each frame
    startXPos = startXPos + 1
```

11.3 Animation: Creating movies from scratch

An example: Needs to be cleaned up for use in general, platform-independent way.

Recipe 48: Animation example

```
def AnimationSimple():
frames = 50
rx = 0
ry = 0
rw = 50
rh = 50
for f in range(1,frames+1):
pic = BlankPicture(500,500)
pic.addOvalFilled(Color(255,0,0),x,y,w,h)
if(f < 10):
pic.writeTo('test0else:
pic.writeTo('test0x = 5 + x
y = 5 + y

def AnimationSimple2():
frames = 100
w = 50
h = 50
#ball postions balls r,g,b,y x and y posistions
rx = 0
ry = 0
bx = 275
by = 225
gx = 275
gy = 275
yx = 225
yy = 275
for f in range(1,frames+1):
pic = BlankPicture(500,500)
if(f < 50):
rx += 5
ry += 5
else:
rx -= 2
ry -= 2
bx += 5
by -= 5
gx += 5
gy += 5
yx -= 5
yy += 5
pic.addOvalFilled(red,rx,ry,w,h)
pic.addOvalFilled(blue,bx,by,w,h)
pic.addOvalFilled(green,gx,gy,w,h)
pic.addOvalFilled(yellow,yx,yy,w,h)
if(f < 10):
pic.writeTo('test00elif(f < 100):
pic.writeTo('test0else:
pic.writeTo('test
```

Exercises

Exercise 46: How would we lighten the *Bladerunner* frame *more*?
Exercise 47: Under what conditions would it not be worth anything to do so?
Exercise 48: Try applying a different manipulation, besides lightening, to frames of a movie.

Chapter 12

Storing Media

Database example will come here.

Part VIII

Isn't there a better way?

Chapter 13

How fast can we get?

When are programs fast, and when are they slow? And why?

13.1 Complexity

Why are the movie programs so slow, while others are so fast?

Here's a rough way of figuring it out: Count the loops.

Look at the normalization recipe (Recipe 11 (page 74)). Do you see two loops? Each of which goes through n samples. We'd say that this *order of complexity* of this recipe is $O(2n)$. As you'll see, the real speed differences have little to do with the constants, so we'll often call this $O(n)$.

Now look at the picture processing code. You'll often see two loops, one working across m pixels and up and down n pixels. We'd call that $O(mn)$. If m and n are near one another, we'd say $O(n^2)$.

Now look at movie processing. l frames, each m by n. That's $O(lmn)$. And if these are close to one another. $O(n^3)$.

13.2 Speed limits

Some things can't be made faster.

Imagine trying to find an optimal arrangement of sounds in a composition/synthesis, or composition of images in a picture. You have to check *every* combination. Let's imagine that you could write a function that will tell you how perfect a picture or sound is.

Let's say that you have 60 sounds you want to arrange and any order is possible, but you want to figure out the best one

Basically, if you have to try every combination of n things, there are 2^n combinations. (You can convince yourself of this pretty easily)

$O(2^60) = 11,52,921,504,606,846,976$

Imagine that you have a 1 Ghz computer (1 billion basic operations per second) – a top of the line processor today

It'll take you 1152921504.606847 seconds to optimize that data

- That's 19,215,358.41011412 minutes
- That's 800,639.933754755 days
- That's 2,193 years
- With Moore's law, in two years, you can do that in only 1,000 years!
- And 60 sounds is a SMALL amount – most songs have many more notes than that

Can we do better? Maybe — can you be satisfied with less than perfect? Can we be smarter than checking EVERY combination? That's part of *heuristics*, a part of what artificial intelligence researchers do.

13.3 Native code versus interpreted code

Why is Photoshop faster than what we're doing? Because Photoshop is written in *native code*.

There is a programming language that computers understand at the level of their wires. It's called *native code* or *machine language*. It's always faster than having the computer interpret a language it doesn't understand natively. Python is an *interpreted language*.

Part IX

Can’t we do this any easier?

Chapter 14

Functional Decomposition

Chapter 15

Recursion: It's functions all the way down

Chapter 16

Objects: Lifting the lid on our media

Remember the *modules* that we saw? Those are akin to *objects*. We'll use *dot notation*.

`sound.writeTo` and `picture.writeTo` are easier than remembering `writeSoundTo` and `writePictureTo`. It's always `writeTo`.

Both colors and pixels understand `getRed`. Makes it easier to work with.

Chapter 17

Other Languages

Bibliography

[Abernethy and Allen, 1998] Abernethy, K. and Allen, T. (1998). *Exploring the Digital Domain: An Introduction to Computing with Multimedia and Networking.* PWS Publishing, Boston.

[Adelson and Soloway, 1985] Adelson, B. and Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on Software Engineering*, SE-11(11):1351–1360.

[Boulanger, 2000] Boulanger, R., editor (2000). *The CSound Book: Perspectives in Synthesis, Sound Design, Signal Processing, and Programming.* MIT Press, Cambridge, MA.

[Bruckman, 2000] Bruckman, A. (2000). Situated support for learning: Storm's weekend with rachael. *Journal of the Learning Sciences*, 9(3):329–372.

[Bruer, 1993] Bruer, J. T. (1993). *Schools for Thought: A Science of Learning in the Classroom.* MIT Press, Cambridge, MA.

[Dodge and Jerse, 1997] Dodge, C. and Jerse, T. A. (1997). *Computer Music: Synthesis, Composition, and Performance.* Schimer:Thomason Learning Inc.

[Greenberger, 1962] Greenberger, M. (1962). *Computers and the World of the Future.* Transcribed recordings of lectures held at the Sloan School of Business Administration, April, 1961. MIT Press, Cambridge, MA.

[Guzdial, 2001] Guzdial, M. (2001). *Squeak: Object-oriented design with Multimedia Applications.* Prentice-Hall, Englewood, NJ.

[Guzdial and Rose, 2001] Guzdial, M. and Rose, K., editors (2001). *Squeak, Open Personal Computing for Multimedia.* Prentice-Hall, Englewood, NJ.

[Harel and Papert, 1990] Harel, I. and Papert, S. (1990). Software design as a learning environment. *Interactive Learning Environments*, 1(1):1–32.

[Ingalls et al., 1997] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. (1997). Back to the future: The story of squeak, a practical smalltalk written in itself. In *OOPSLA'97 Conference Proceedings*, pages 318–326. ACM, Atlanta, GA.

[Resnick, 1997] Resnick, M. (1997). *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds.* MIT Press, Cambridge, MA.

[Roads, 1996] Roads, C. (1996). *The Computer Music Tutorial.* MIT Press, Cambridge, MA.

Index