CHAPTER 1

# Introduction to Computer Science and Media Computation


**1.1** What Is Computer Science About?
**1.2** Programming Languages
**1.3** What Computers Understand
**1.4** Media Computation: Why Digitize Media?
**1.5** Computer Science for Everyone


## C *hapter Learning Objectives*

- *To explain what computer science is about and what computer scientists are concerned with.*
- *To explain why we digitize media.*
- *To explain why it's valuable to study computing.*
- *To explain the concept of an* **encoding.**
- **To explain the basic components of a computer.**

## 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 we'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 this point is only of importance to computer scientists. The important point overall is that a computer science recipe defines *exactly* what has to be done.**

**If you're a biologist who wants to describe how migration works or how DNA replicates, then being able to write a recipe that specifies *exactly* what happens, in terms that can be completely defined and understood, is *very* useful. The same is true if you're a chemist who wants to explain how equilibrium is reached in a reaction. A factory manager can define a machine-and-belt layout and even test how it works before physically moving heavy things into position using computer programs. A recipe that can run on a computer is called a *program* . Being able to exactly define tasks and/or simulate events is a major reason why computers have radically changed so much of how science is done and understood.**

**It may sound funny to call *programs* 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 recipes for some programs, like the one that keeps track of Visa/MasterCard records, have literally millions of steps!). The term software means a collection of computer programs (recipes) that accomplish a task.**

■ **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 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, 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 something like meters per second?**

**Sometimes, in certain special situations—on a ship at sea, for instance—the 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 know that there are recipes that can't be written. For example, you can't write a recipe that can absolutely tell whether some other recipe will actually work. How about *intelligence*? Can we write a recipe such that a computer following it would actually be *thinking* (and how would you tell if you got it right)? Computer scientists 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 restaurant critics for a newspaper. Some of these are human–computer interface specialists who worry about whether people like how the recipes work ("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 barbecue, 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 phenomenon 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 a recipe can be changed without changing the result dramatically. You can always increase all the units by a multiplier (say, double) 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 and then sauté them, the reverse order will probably not work well (Figure 1.1).**

**The same holds 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.**

## 1.2 Programming Languages

**Computer scientists write a recipe in a programming language (Figure 1.2). 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 or 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 human languages, like English or Spanish? The problem is that natural languages evolved the way they did to enhance communications between very smart beings, humans. As we'll explain 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**

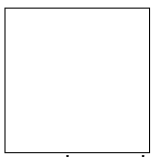

**FIGURE 1.1**: A cooking recipeÐyou can always double the ingredients, but throwing in an extra cup of flour won't cut it, and don't try to brown the chicken *after* adding the tomato sauce!.

**programming language *Lisp* (and 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 (as a natural consequence) few who know it. If you want to hire a hundred programmers to work on your project, it will be 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 it). Python is a fairly popular programming language, used very often for Web and media programming. The Web search engine *Google* uses Python. The media company *Industrial Light & Magic* also uses Python. A list of companies using Python is available at `http://www.python.org/psa/Users.html`. Python is easy to learn, easy to read, very flexible, but not**

**Python/Jython**

```
def hello():
  print "Hello World"
```

**Java**

```
class HelloWorld {
  static public void main( String args[] ) {
    System.out.println( "Hello World!" );
  }
}
```

**C++**

```
#include <iostream.h>

main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

**Scheme**

```
(define helloworld
        (lambda ()
                (display "Hello World")
                (newline)))
```

**FIGURE 1.2**: Comparing programming languages: A common simple programming task is to print the words "Hello, World!" to the screen.

**very efficient. The same algorithm coded in C and in Python will probably be faster in C.**

**The version of Python used in this book is called Jython (`http://www.jython.org`). Python is normally implemented in the programming language C. Jython is Python implemented in *Java*—this means that Jython is actually a program written in Java. Jython lets us do multimedia that will work across multiple computer platforms. Jython is a real programming language that can be used for serious work. You can download a version of Jython for your computer from the Jython Web site that will work for all kinds of purposes. We will be using Jython in this book through a special programming *environment* called JES (*Jython Environment for Students*) that has been developed to make it easier to program in Jython. But anything you can do in JES, you can also do in normal Jython—and most programs that you write in Jython will also work in Python.**

**Here is an explanation of some of the important terms that we'll be using in this book:**

- **A program is a description in a programming language of a process that achieves some result that is useful to someone. A program can be small (like one that implements a calculator) or huge (like one your bank uses to track all of its accounts).**
- **An algorithm (in contrast) is a description of a process apart from any programming language. The same algorithm may be implemented in many different languages in many different ways in many different programs—but they would all be the same process if we're talking about the same algorithm.**

**The term recipe, as used in this book, describes programs or portions of programs that do something. We're going to use the term recipe to emphasize the *pieces of a program that achieve a useful* media-related *task*.**

## 1.3 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 not really correct. Computers use encodings of numbers. Computers are electronic devices that react to voltages on wires. Each wire 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. We group these wires (bits) into sets. A set of 8 bits is called a byte. So, from a set of eight wires (a byte), we have a pattern of eight 0's and 1's, e.g., 01001010. Using the binary number system, we can interpret this byte as a number (Figure 1.3). That's where we come up with the claim that a computer knows about numbers.**[1]

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

**A computer has a memory filled with bytes. Everything that a computer is working with at a given instant is stored in its memory. This means that everything 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 e-mail.**

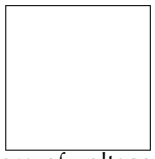

**FIGURE 1.3**: Eight wires with a pattern of voltages is a byte, which is interpreted as a pattern of eight 0's and 1's, which is interpreted as a decimal number.

**A computer can do lots of things with numbers. It can add them, subtract them, multiply them, divide them, sort them, collect them, duplicate 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, it looks like the computer is a kind of fancy calculator, and that's certainly why it was invented. The first use of a computer was to calculate projectile trajectories during World War II ("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 …"). Modern computers can do billions of calculations per second. But what makes the computer 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 interpret as numbers.**

**If one of these bytes is interpreted as the number** 65 **, it could simply 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 the 65 appears in a collection of other numbers that we're interpreting as text, and it'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 in 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 the same 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 this way, it slows the computer down some. But the amazing thing about computers is that they're *amazingly* fastÐand 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 computers running Windows operating systems), claimed 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. This means that computers keep getting smaller, faster, and cheaper. This law has held true for decades.**

**Computers today can execute literally *billions* of recipe steps per second. They can hold in memory literally encyclopedias of data! They never get tired or bored. Search a million customers for an individual 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. In this book you will write recipes that manipulate**

**images, sounds, text, and even other recipes. This is possible because everything in the computer is represented digitally, even recipes. By the end of the book you will have written recipes to implement digital video special effects, that create web pages in the same way that Amazon and E-bay does, and that filter images like PhotoShop.**

## 1.4 Media Computation: Why Digitize Media?

**Let's consider an encoding that would be appropriate for pictures. Imagine that pictures are made up of little dots. That's not hard to imagine: Look really closely at your monitor or at a TV screen and you will see that your images are *already* made up of little dots. Each of these dots is a distinct color. Physics tells us 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 3.**

**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 very 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 (twice that for stereo). 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 encodings of this kind 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 and 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?) 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 our ears and most measurements, a CD (which *is* digitized sound) sounds just the same, or maybe clearer. Digital cameras at high-enough resolutions produce photograph-quality pictures.**

**Why would you want to digitize media? Because then the media will be 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.**

**When we manipulate media we need to be respectful of the author's digital rights. Modifying images and sounds for educational purposes is allowed under fair use. However, sharing or publishing manipulated images or sounds could infringe on the owner's copyright.**

## 1.5 Computer Science for Everyone

**Why should you learn about computer science by writing programs that manipulate media? 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 by manipulating media?**

**Most professionals today 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 sections provide answers to these questions.**

### 1.5.1 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 that can't be said in software from Adobe, Microsoft, Apple, and the rest. Or what if you want to say something in a way they don't support? If you know how to program, even if it would take you *longer* to do it yourself, you have the freedom to manipulate the media your way.**

**What about learning these tools in the first place? In all our years working with computers, we have seen many types of software come and go as *the* package for drawing, painting, word-processing, video editing, and so on. You can't learn just a single tool and expect to be able to use it for 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.**

### 1.5.2 It's About Process

**In 1961, Alan Perlis gave a talk at MIT in which he argued that computer science, and programming explicitly, should be part of a liberal education [17]. Perlis is an important figure in the field of computer science. The highest award in computer science 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's argument can be 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 stated earlier in this chapter, 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 for everyone. Using a computer to automate processes has changed every profession.**

**More recently, Jeannette Wing has argued that everyone should learn computational thinking [34]. She views the types of skills taught in computing as critical skills for all students. This is what Alan Perlis predicted in that automating computation would change the way we learn about our world.**

## Problems

**1.1** *Every* profession uses computers today. Use a Web browser and a search engine like Google to find sites that relate your field of study with computer science or computing or computation. For example, search for "biology computer science" or "management computing."

**1.2** Find an ASCII table on the Web: a table listing every character and its corresponding numeric representation. Write down the sequence of numbers whose ASCII values make up your name.

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

**1.4** Consider the representation for pictures described in Section 1.4, 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 640 by 480 picture, a common picture size on the Web? How many bytes does it take to represent a 1024 by 768 picture, a common screen size? (What do you think is meant now by a "three megapixel" camera?)

**1.5** One bit can represent 0 or 1. With two bits you have four possible combinations 00, 01, 10, and 11. How many different combinations can you make with four bits or eight bits (one byte)? Each combination can be used to represent a binary number. How many numbers can you represent with 2 bytes (16 bits). How many numbers can you represent with four bytes?

***1.6** How can you represent a *floating point number* in terms of bytes? Do a search on the Web for "floating point" and see what you find.

**1.7** Look up Alan Kay and the *Dynabook* on the Web. What does he have to do with media computation?

**1.8** Look up Grace Hopper on the Web? How did she contribute to programming languages?

**1.9** Look up Philip Emeagwali on the Web? What computing prize did he win?

**1.10** Look up Alan Turing on the Web. What does he have to do with our notion of what a computer can do and how encodings work?

**1.11** Look up the Harvard computers on the web. What did they contribute to astronomy?

**1.12** Look up Adele Goldberg on the Web. How did she contribute to programming languages?

**1.13** Look up Kurt Gödel on the Web. What amazing things did he do with encodings?

**1.14** Look up Ada Lovelace on the Web. What amazing things did she do before the first mechanical computer was built?

**1.15** Look up Claude Shannon on the Web. What do he do for his master's thesis?

**1.16** Look up Richard Tapia on the Web. What has he done to encourage diversity in computing?

**1.17** Look up Frances Allen on the Web. What computing prize did she win?

**1.18** Look up Mary Lou Jepsen on the Web. What new technology is she working on?

**1.19** Look up Ashley Qualls on the Web. What did she create that is worth a million dollars?

**1.20** Look up Marissa Mayer on the Web. What does she do?

## To Dig Deeper

**James Gleick's book *Chaos* describes more on emergent properties—how small changes can lead to dramatic effects, and the unintended impacts of designs because of difficult-to-foresee interactions.**

**Mitchel Resnick's book *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds* [33] describes how ants, termites, and even traffic jams and slime molds can be described pretty accurately with hundreds or thousands of very small processes (programs) running and interacting all at once.**

***Exploring The Digital Domain* [3] is a wonderful introductory book on computation with lots of good information about digital media.**