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

Chapter 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 or computers do things, how we specify what we do, and how we specify what the stuff is that we're processing. That's a pretty dry definition. Let's try a metaphorical one.

Computer Science Idea: Computer Science Is the Study of Recipes

"Recipes" here are a special kind—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.

More formally, computer scientists study *algorithms* which are step-by-step procedures to accomplish a task. Each step in an algorithm is something that a computer already knows how to do (e.g., add two small integer numbers) or can be taught how to do (e.g., adding larger numbers including those with a decimal point). A recipe that can run on a computer is called a *program*. A program is a way to communicate an algorithm in a representation that a computer can execute. ■

To use our metaphor a bit more—think of an algorithm as the step-by-step way that your grandmother made her secret recipe. She always did it the same way, and had a

reliably great result. Writing it down so that you can read it and do it later is like turning her algorithm into a program for you. You *execute* the recipe by *doing* it—following the recipe step-by-step in order to create something the way that your grandmother did. If you give the recipe to someone else who can read the language of the recipe (maybe English or French), then you have communicated that process to that other person, and the other person can similarly execute the recipe to make something the way that your grandmother did.

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**. 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.

In fact, if you *can't* write a recipe for some process, maybe you don't really understand the process, or maybe the process can't actually work the way that you are thinking about it. Sometimes, trying to write the recipe is a test in itself. Now, sometimes you can't write the recipe because the process is one of the few that cannot be executed by a computer. We will talk more about those in Chapter 14.

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 egg whites from yolks, you realize that knowing the right way to do it makes a world of difference. Computer science theoreticians think 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), or even use the least amount of energy (which is important when running on low-power devices like cell phones). *How* a recipe works, completely apart from how it's written (e.g., in a program), is called the study of algorithms. Software engineers think about how large groups can put together recipes that still work. (Some programs, like the ones that keep track of credit card transactions, 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. Or we may invent new kinds of units, like a unit that represents a whole other program or a computer, or

a network like your friends and your friends' friends in Facebook. 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) and figuring out how to access the data quickly 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 focus on whether people *like* what the recipes produce, almost like restaurant critics for a newspaper. Some of these are **human–computer interface** specialists who worry about whether people can understand and make use of the recipes ("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 certain 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 noncomputational 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. There are computer scientists today who study how the Web allows for new kinds of interactions, particularly in large groups (like Facebook or Twitter). Computer scientists who study *social computing* are interested in how these new kinds of interactions work and the characteristics of the software that are most successful for promoting useful social interactions.

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

CHICKEN CACCIATORE

3 whole, boned chicken breasts
1 medium onion, chopped
1 tbsp chopped garlic
2 tbsp and later ¼ c olive oil
1 ½ c flour
¼ c Lawry's seasoning salt
1 bell pepper, chopped (optional) any color
1 (28 oz) can chopped tomatoes
1 (15 oz) can tomato sauce
1 (6.5 oz) can mushrooms
1 (6 oz) can tomato paste
½ of (26 oz) jar of spaghetti sauce
3 tbsp Italian seasoning
1 tsp garlic powder (optional)

Cut up the chicken into pieces about 1 inch square. Saute the onion and garlic until the onion is translucent. Mix the flour and Lawry's salt. You want about 1:4–1:5 ratio of seasoning salt to flour and enough of the whole mixture to coat the chicken. Put the cut up chicken and seasoned flour in a bag, and shake to coat. Add the coated chicken to the onion and garlic. Stir frequently until browned. You'll need to add oil to keep from sticking and burning; I sometimes add up to ¼ cup of olive oil. Add the tomatoes, sauce, mushrooms, and paste (and the optional peppers, too). Stir well. Add the Italian seasoning. I like garlic, so I usually add the garlic powder, too. Stir well. Because of all the flour, the sauce can get too thick. I usually cut it with the spaghetti sauce, up to ½ jar. Simmer 20–30 minutes.

FIGURE 1.1
A cooking recipe—you 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!

soda. The order matters. If you're supposed to brown the chicken and then add tomato sauce, you won't get the same result if you add tomato sauce and then (somehow) try to brown the chicken (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 safely changed, 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 Scala. Some 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 to experts 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

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.

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 video game like *Mario Kart* or *Minecraft* or *Call of Duty* 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 and *people* use these languages. 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 related languages like Scheme, Racket, 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. 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** (visit `http://www.python.org` for more information on it). Python is a 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://css.dzone.com/articles/best-python-companies-work`. Python is easy to learn, easy to read, very flexible, but not as efficient as other programming languages. The same algorithm coded in C and in Python will probably be faster in C. Python is a good language for writing programs that work within an application, like the image manipulation language GIMP (`http://www.gimp.org`) or the 3D content creation tool Blender (`http://www.blender.org`).

The version of Python used in this book is called **Jython** (`http://www.jython.org`).[1] Jython is a form of Python that is particularly effective for programming in multimedia that will work across multiple computer platforms. You can download a version of Jython for your computer from the Jython Web site that will work for all kinds of purposes. Most programs written for Jython will work without change in Python.

In this book, we will describe how to program in Jython using a programming *environment* called **JES** (*Jython Environment for Students*) that has been developed to make it easier to program in Jython. JES has some features for working with media, like viewers for sounds and images. JES also has embedded in it some special functions for manipulating digital media that are made available to you without you doing anything special. Anything you can do in JES, you can also do in normal Jython, though you will have to explicitly include the special libraries.

Media Computation special libraries work with other environments and other forms of Python, too.

- Pythy developed by Stephen Edwards and his colleagues at Virginia Tech is a browser-based programming environment.[2] All programming occurs through a Web browser with Pythy, and programs and media are stored in the cloud. Programs that work in JES will work in Pythy.
- A team led by Paul Gries at the University of Toronto has implemented portions of the special media libraries in JES to work in other forms of Python.[3]
- The books at `http://www.interactivepython.org/` by Brad Miller at Luther College support manipulation of images in Python in all of their ebooks.

Let's revisit here two of the most 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).

[1] Python is often implemented in the programming language C. Jython is Python implemented in *Java*—this means that Jython is actually a program written in Java.

[2] See `https://github.com/web-cat/pythy` for information on installing Pythy.

[3] See `https://code.google.com/p/pygraphics/` for information on the Pygraphics library.

- An **algorithm** (in contrast) is a description of a process in a step-by-step manner, not tied to 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.

Computer Science Idea: Computer Science is about People

A famous computer scientist, Edsger W. Dijkstra, once said that, "Computer Science is no more about computers than astronomy is about telescopes." Computer science is about people. People think about process in all those different ways (from a data perspective, to issues of human–computer interfaces, to artificial intelligence). People use programming languages, and they prefer different ways of communicating and thinking, so different languages result. People make the programs and the languages. Most decisions in computer science are not about computers. Most decisions in computer science are about people.

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, for example, 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.[4]

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 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 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. One of the first uses 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

[4]We'll talk more about this level of the computer in the chapter *Speed*.

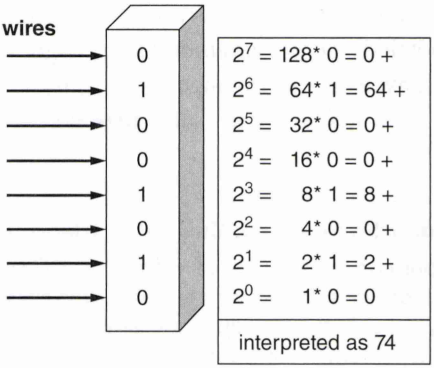

FIGURE 1.3
Eight wires with a pattern of voltages is a byte, which is interpreted as a pattern of eight 0's and 1's, which in turn is interpreted as a decimal number.

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 bottom-most level, the computer *only* "knows" voltages, which we interpret as numbers. Encodings let us forget about lower-level details. Encodings are an example of *abstraction*, which give us new concepts to use which allow us to ignore other details.

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 a bit. But the powerful 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) 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 cardholder? 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. We would not be able to do media computation if the media were not represented digitally. 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 eBay do, 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 a 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 4.