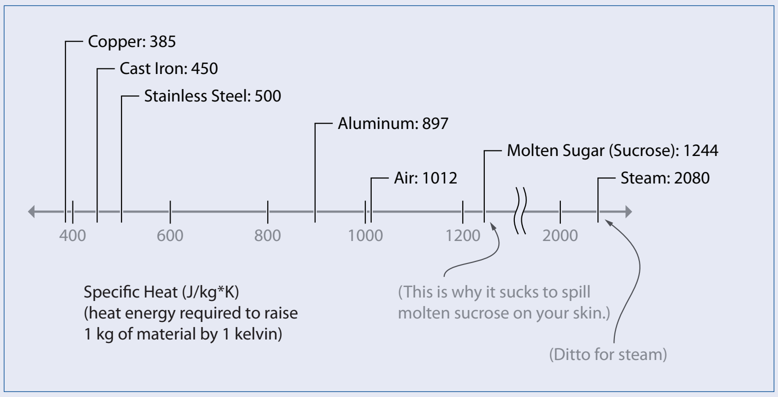


kilogram of cast iron 1°C versus a kilogram of aluminum, because of how the materials are structured at the atomic level. How do common metals in pans compare in terms of specific heat?

Cast iron has a lower specific heat than aluminum. It takes roughly twice as much energy (897 J/kg*K versus 450 J/kg*K) to heat the same amount of aluminum up to the same temperature, and because energy doesn't just disappear (first law of thermodynamics), this means that a kilogram of aluminum will actually give off *more* heat than a kilogram of cast iron as it cools (e.g., when you drop that big steak onto the pan's surface).

It's not just the thermal conductivity or specific heat of the metal that matters, though; the mass of the pan is critical. I always sear my steak in my cast iron pan. It weighs 7.7 lbs / 3.5 kg, as opposed to 3.3 lbs / 1.5 kg in the case of my aluminum pan, so it has more heat energy to give off. When searing, pick a pan that has the highest value of *specific heat * mass*, so that once it's hot, it won't drop in temperature as much when you add the food.

There are a few other factors you should consider when picking a pan. Cast iron and aluminum react with acids, so pans made of those materials shouldn't be used for simmering tomatoes or other acidic items. Nonstick pans shouldn't be heated above 500°F / 260°C. And then there are cases where the pan isn't the primary source of heat for cooking: when boiling or steaming, the water provides the heat transfer, so the material used in making the pan isn't important. Likewise, if you're using an ultra-high-BTU burner (like the 60,000-BTU burners used in wok cooking), the pan isn't a heat sink so heat capacity isn't important.

What's the deal with *cladded* metals? You know, pans with copper or aluminum cores, encased in stainless steel or some other metal? (*Clad* = to encase with a covering.) These types of pans are a solution to two goals: avoiding hot spots by evening out heat quickly (by using aluminum or copper), and using a nonreactive surface (typically stainless steel, although nonstick coatings also work) so that the food doesn't chemically react with the pan.

Finally, if you're buying a pan and can't decide between two otherwise identical choices, go for the one that has oven-safe handles. Avoid wood, and make sure the handles aren't so big that they prevent popping the pan in the oven.

Measuring cups and scales

In addition to the common items used for measuring (e.g., measuring cups and spoons), I strongly recommend purchasing a kitchen scale. If you will be following any of the recipes from this book using hydrocolloids or other food additives (see Chapter 6), it is practically required. You might not use it every day (or even every week), but there is no substitute for it when you need one.

You can pour ingredients directly into a mixing bowl by weight, skipping the need for measuring cups.

You will obtain better accuracy when measuring by weight. Dry ingredients such as flour can become compressed, so the amount of flour in "1 cup" can vary quite a bit due to the amount of pressure present when it's packed (see the sidebar "Weight Versus Volume: The Case for Weight"). Also, it is easier to precisely measure weight than volume. Because much of cooking is about controlling chemical reactions based on the ratio of ingredients (say, flour and water), changes in the ratio will alter your results, especially in baking. Weighing ingredients also allows you to load ingredients serially: add 390 grams of flour, hit tare; 300 grams of water, hit tare; 7 grams of salt, hit tare; 2 grams of yeast, mix, let rest for 20 hours, and you've got no-knead bread. (See the interview with Martin Lersch on page 224 in Chapter 5 for baking instructions.)

Use a high-precision scale when working with food additives.

When choosing a scale, look for the following features:

- A digital display, showing weights in grams and ounces, that has a tare function for zeroing out weight
- A flat surface on which you can place a bowl or dish (avoid scales that have built-in bowls)
- A scale that is capable of measuring up to at least 5 lbs or 2.2 kg in 0.05 oz or 1g increments

If you plan on following any "molecular gastronomy / modernist cuisine" recipes that use chemicals, you'll need to pick up a *high-precision scale* that measures in increments of 0.1 gram or finer. I use an American Weigh Scale AMW-100.

Spoons & co.

Few things symbolize cooking more than a spoon, and for good reason: stirring, tasting, adjusting the seasoning, stirring some more, and tasting again would be virtually impossible without a good spoon! I prefer the wooden variety. In an age of technology and modern plastics, there's just something comforting about a wooden spoon. Look for one that has a straight end, as opposed to a traditional spoon shape, because the straight edge is useful for scraping the inside corners and bottom of a pan to release fond. When it comes to cleaning them, I run mine through the dishwasher. True, it's bad for the wood, but I find it easier and don't mind buying a new one every few years.

Weight Versus Volume: The Case for Weight

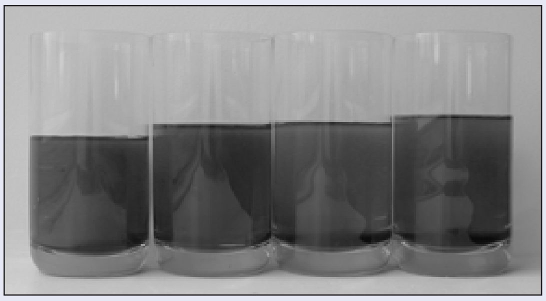
How much of a difference does it *really* make to weigh your flour? To find out, I asked friends to measure out 1 cup of all-purpose flour and then weigh it. Ten cups later, the gram weights were in: 124, 125, 131, 133, 135, 156, 156, 158, 162, and 163. That's a whopping 31% difference between the lowest and highest measurements.

How much flour is in a cup? Depends on whether you pack it in tight (on left: 1 cup at 156 grams, then sifted) or keep it loose (on right: 1 cup at 125 grams, then sifted).

Even if you could perfectly measure the same weight with every cup, you still might end up using a different amount than what a recipe calls for. The average weight of the 10 samples above is 144 grams. The United States Department of Agriculture defines 1 cup of flour as 125 grams; Wolfram|Alpha (*http://www.wolframalpha.com*) gives 137 grams. And the side of the package of flour in my kitchen? 120 grams.

The upshot? You'll get better results by weighing ingredients, especially when baking. A cup might not be a cup, but 100 grams will always be 100 grams. Clearly, weight is the way to go.

But what about wet measurements—measurements of things that don't compress? While you're not going to see the same variability, you can still end up with a fair amount of skew just based on the accuracy of the measuring device. The following image shows what four different methods for measuring 1 cup of liquid yielded.

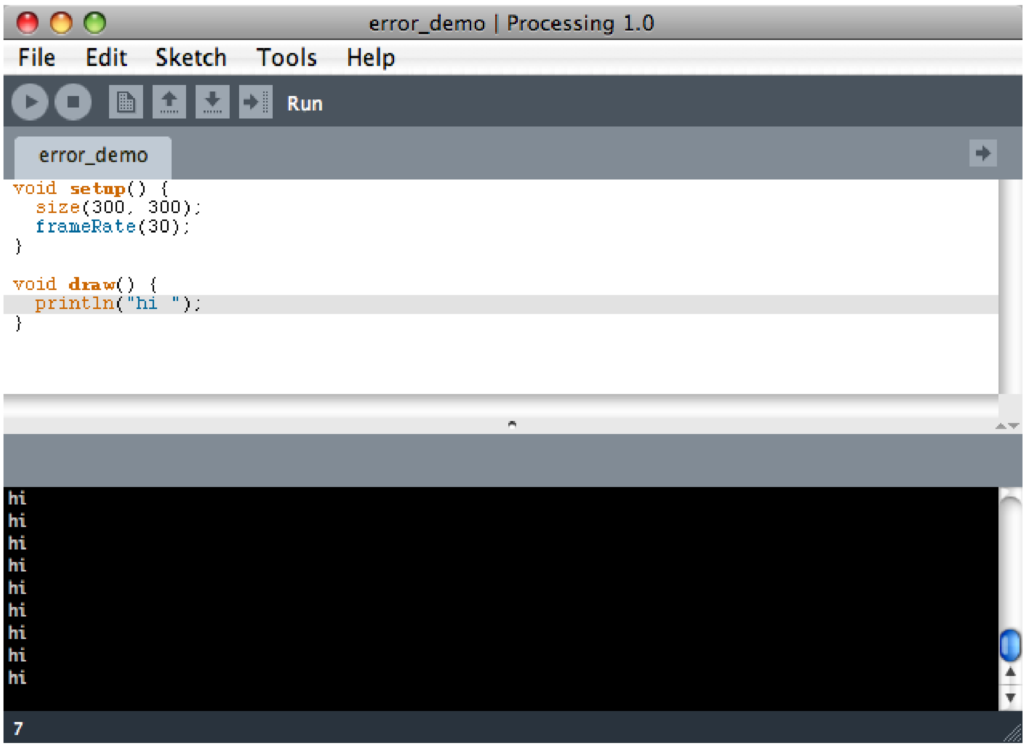


212 grams
Tablespoon
(16 tablespoons = 1 cup)

225 grams
Liquid measuring cup

232 grams
Dry measuring cup

237 grams
Digital scale



```
void setup() {
  size(300, 300);
  frameRate(30);
}

void draw() {
  println("hi ");
}
```

```
hi
hi
hi
hi
hi
hi
hi
hi
hi
```

```
7
```

Figure 3-2. The Console window in an application

The draw() Method

The `draw()` method is where the drawing of the application happens, but it can be much more than that. The `draw()` method is the heartbeat of your application; any behavior defined in this method will be called at the number of times per second specified as the frame rate of your application.

A simple example of a `draw()` method can be seen in Example 3-3.

Example 3-3. methods.pde

```
void draw() {
    println("hi");
}
```

Assuming that the frame rate of your application is 30 times a second, the message `"hi"` will print to the Console window of the Processing IDE 30 times a second. That's not very exciting, is it? But it demonstrates what the `draw()` method is: the definition of the behavior of any processing application at a regular interval determined by the frame rate, after the application runs the `setup()` method.

Example 3-4 is a slightly more interesting example dissected.

Example 3-4. expanding.pde

```
int x = 0;

void setup() {
    size(300, 300);
}

void draw() {
    // make x a little bit bigger
    x += 2;
    // draw a circle using x as the height and width of the circle
    ellipse(150, 150, x, x);
    // if x is too big, we can't see it in our window, so put it back
    // to 0 and start over
    if(x > 300) {
        x = 0;
    }
}
```

First things first—you're making an `int` variable, `x`, to store a value:

```
int x = 0;
```

Since `x` isn't inside a method, it's going to exist throughout the entire application. That is, when you set it to 20 in the `draw()` method, then the next time the `draw()` method is called the value of `x` is still going to be 20. This is important because it lets you gradually animate the value of `x`.

This refers to the idea of *scope*; if that concept isn't ringing any bells for you, review Chapter 2.

To set up the application using the `setup()` method, simply set the size of the window so that it's big enough. Nothing too interesting there, so you can skip right to the `draw()` method:

```
void draw() {
```

Each time you call `draw()`, you're going to make this number bigger by 2. You could also write `x = x+2;`, but the following is simpler and does the same thing:

```
    x += 2;
```

Now that you've made `x` a little bit bigger, you can use it to draw a circle into the window:

```
ellipse(150, 150, x, x);
```

Look ahead in this chapter to the section "The Basics of Drawing with Processing" on page 63 for more information about the `ellipse()` method.

If the value of `x` is too high, the circle will be drawn too large for it to show up correctly in your window (300 pixels); you'll want to reset `x` to 0 so that the circles placed in the window begin growing again in size:

```
        if(x > 300) {
            x = 0;
        }
    }
```

In Figure 3-3, you can see the animation about halfway through its cycle of incrementing the `x` value and drawing gradually larger and larger circles.

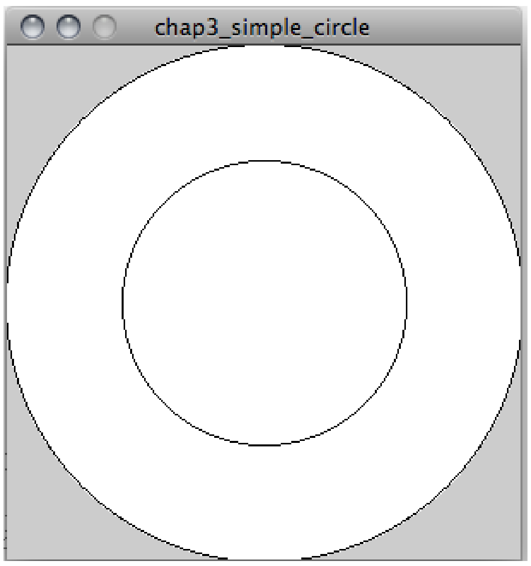


Figure 3-3. The demo application drawing circles

The `draw()` method is important because the Processing application uses it to set up a lot of the interaction with the application. For instance, the `mousePressed()` method and the `mouseMove()` methods that are discussed in the section "Capturing Simple User Interaction" on page 70 will not work without a `draw()` method being defined. You can imagine that the `draw()` method tells the application that you want to listen to whatever happens with the application as each frame is drawn. Even if nothing is between the brackets of the `draw()` method, generally you should always define a `draw()` method.

Every Processing program has two main routines, setup() and draw(). setup() happens once at the beginning of the program. It's where you set all your initial conditions, like the size of the applet window, initial states for variables, and so forth. draw() is the main loop of the program. It repeats continuously until you close the applet window.

In order to use variables in Processing, you have to declare the variable's data type. In the preceding program, the variables redValue, greenValue, and blueValue are all float types, meaning that they're floating decimal-point numbers. Other common variable types you'll use are ints (integers), booleans (true or false values), Strings of text, and bytes.

Like C, Java, and many other languages, Processing uses C-style syntax. All functions have a data type, just like variables (and many of them are the void type, meaning that they don't return any values). All lines end with a semicolon, and all blocks of code are wrapped in curly braces. Conditional statements (if-then statements), for-next loops, and comments all use the C syntax as well. The preceding code illustrates all of these except the for-next loop.

Here's a typical for-next loop. Try this in a sketch of its own (to start a new sketch, select New from Processing's File menu).

```
for (int myCounter = 0; myCounter <=10; myCounter++) {
  println(myCounter);
}
```

BASIC users: **If you've never used a C-style for-next loop, it can seem forbidding. What this bit of code does is establish a variable called** myCounter**. As long as a number is less than or equal to 10, it executes the instructions in the curly braces.** myCounter++ **tells the program to add one to** myCounter **each time through the loop. The equivalent BASIC code is:**

```
for myCounter = 0 to 10
  Print myCounter
next
```

Processing is a fun language to play with because you can make interactive graphics very quickly. It's also a simple introduction to Java for beginning programmers. If you're a Java programmer already, you can include Java directly in your Processing programs. Processing is expandable through code libraries. You'll be using two of the Processing code libraries frequently in this book: the serial library and the networking library.

For more on the syntax of Processing, see the language reference guide at www.processing.org. To learn more about programming in Processing, check out **Processing: A Programming Handbook for Visual Designers and Artists**, by Casey Reas and Ben Fry (MIT Press), the creators of Processing, or their shorter book, **Getting Started with Processing** (O'Reilly). Or, read Daniel Shiffman's excellent introduction, **Learning Processing** (Morgan Kaufmann). There are dozens of other Processing books on the market, so find one whose style you like best.

Remote-Access Applications

One of the most effective debugging tools you'll use when making the projects in this book is a command-line remote-access program, which gives you access to the command-line interface of a remote computer. If you've never used a command-line interface before, you'll find it a bit awkward at first, but you get used to it pretty quickly. This tool is especially important when you need to log into a web server, because you'll need the command line to work with PHP scripts that will be used in this book.

Most web hosting providers are based on Linux, BSD, Solaris, or some other Unix-like operating system. So, when you need to do some work on your web server, you may need to make a command-line connection to your web server.

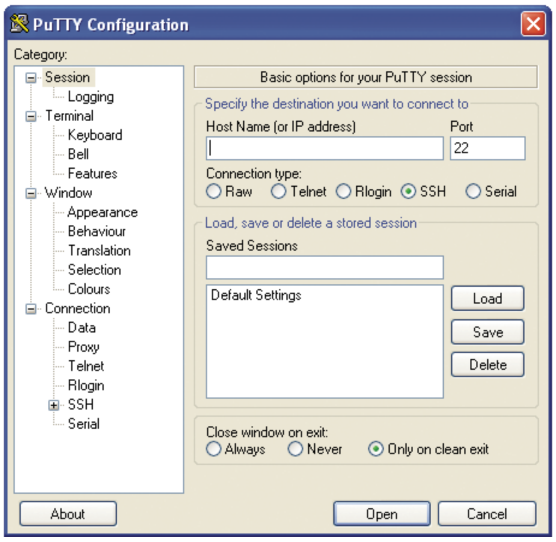
NOTE: **If you already know how to create PHP and HTML documents and upload them to your web server, you can skip ahead to the "PHP" section.**

Although this is the most direct way to work with PHP, some people prefer to work more indirectly, by writing text files on their local computers and uploading them to the remote computer. Depending on how restrictive your web hosting service is, this may be your only option (however, there are many inexpensive hosting companies that offer full command-line access). Even if you prefer to work this way, there are times in this book when the command line is your only option, so it's worth getting to know a little bit about it now.

On Windows computers, there are a few remote access programs available, but the one that you'll use here is called PuTTY. You can download it from www.puttyssh.org. Download the Windows-style installer and run it. On Mac OS X and Linux, you can use OpenSSH, which is included with both operating systems, and can be run in the Terminal program with the command ssh.

Before you can run OpenSSH, you'll need to launch a terminal emulation program, which gives you access to your Linux or Mac OS X command line. On Mac OS X, the program is called Terminal, and you can find it in the **Utilities** subdirectory of the **Applications** directory. On Linux, look for a program called xterm, rxvt, Terminal, or Konsole.

NOTE: ssh **is a more modern cousin of a longtime Unix remote-access program called** telnet. ssh **is more secure; it scrambles all data sent from one computer to another before sending it, so it can't be snooped on en route.** telnet **sends all data from one computer to another with no encryption. You should use** ssh **to connect from one machine to another whenever you can. Where** telnet **is used in this book, it's because it's the only tool that will do what's needed for the examples in question. Think of** telnet **as an old friend: maybe he's not the coolest guy on the block, maybe he's a bit of a gossip, but he's stood by you forever, and you know you can trust him to do the job when everyone else lets you down.** X



Host Name (or IP address):
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Close window on exit: Always Never [✓ Only on clean exit]

Figure 1-3
The main PuTTY window.

Making the SSH Connection

Mac OS X and Linux

Open your terminal program. These Terminal applications give you a plain-text window with a greeting like this:

```
Last login: Wed Feb 22 07:20:34 on ttyp1
ComputerName:~ username$
```

Type ssh username@myhost.com at the command line to connect to your web host. Replace username and myhost.com with your username and host address.

Windows

On Windows, you'll need to start up PuTTY (see Figure 1-3). To get started, type myhost.com (your web host's name) in the Host Name field, choose the SSH protocol, and then click Open.

The computer will try to connect to the remote host, asking for your password when it connects. Type it (you won't see what you type), followed by the Enter key.

Using the Command Line

Once you've connected to the remote web server, you should see something like this:

```
Last login: Wed Feb 22 08:50:04 2006 from 216.157.45.215
[userid@myhost ~]$
```

Now you're at the command prompt of your web host's computer, and any command you give will be executed on that computer. Start off by learning what directory you're in. To do this, type:

```
pwd
```

which stands for "print working directory." It asks the computer to list the name and pathname of the directory in which you're currently working. (You'll see that many Unix commands are very terse, so you have to type less. The downside of this is that it makes them harder to remember.) The server will respond with a directory path, such as:

```
/home/igoe
```

This is the home directory for your account. On many web servers, this directory contains a subdirectory called **public_html** or **www**, which is where your web files belong. Files that you place in your home directory (that is, outside of **www** or **public_html**) can't be seen by web visitors.

NOTE: You should check with your web host to learn how the files and directories in your home directory are set up.

To find out what files are in a given directory, use the list (ls) command, like so:

```
ls -l .
```

NOTE: The dot is shorthand for "the current working directory." Similarly, a double dot is shorthand for the directory (the parent directory) that contains the current directory.

The -l means "list long." You'll get a response like this:

```
total 44
drwxr-xr-x  13 igoe users 4096 Apr 14 11:42 public_html
drwxr-xr-x   3 igoe users 4096 Nov 25  2005 share
```

This is a list of all the files and subdirectories of the current working directories, as well as their attributes. The first column lists who's got permissions to do what (read, modify, or execute/run a file). The second lists how many links there are to that file elsewhere on the system; most of the time, this is not something you'll have much need for. The third column tells you who owns it, and the fourth tells you the group (a collection of users) to which the file belongs. The fifth lists its size, and the sixth lists the date it was last modified. The final column lists the filename.

In a Unix environment, all files whose names begin with a dot are invisible. Some files, like access-control files that you'll see later in the book, need to be invisible. You can get a list of all the files, including the invisible ones, using the –a modifier for ls, this way:

```
ls -la
```

To move around from one directory to another, there's a "change directory" command, cd. To get into the **public_html** directory, for example, type:

```
cd public_html
```

To go back up one level in the directory structure, type:

```
cd ..
```

To return to your home directory, use the ~ symbol, which is shorthand for your home directory:

```
cd ~
```

If you type cd on a line by itself, it also takes you to your home directory.

If you want to go into a subdirectory of a directory, for example the **cgi-bin** directory inside the **public_html** directory, you'd type cd public_html/cgi-bin. You can type the absolute path from the main directory of the server (called the root) by placing a **/** at the beginning of the file's pathname. Any other file pathname is called a relative path.

To make a new directory, type:

```
mkdir directoryname
```