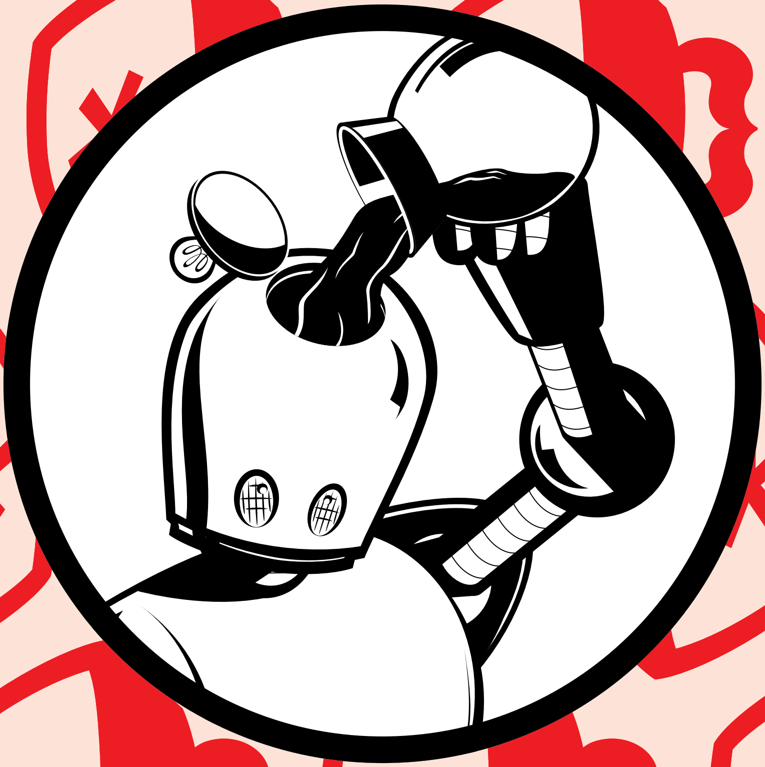


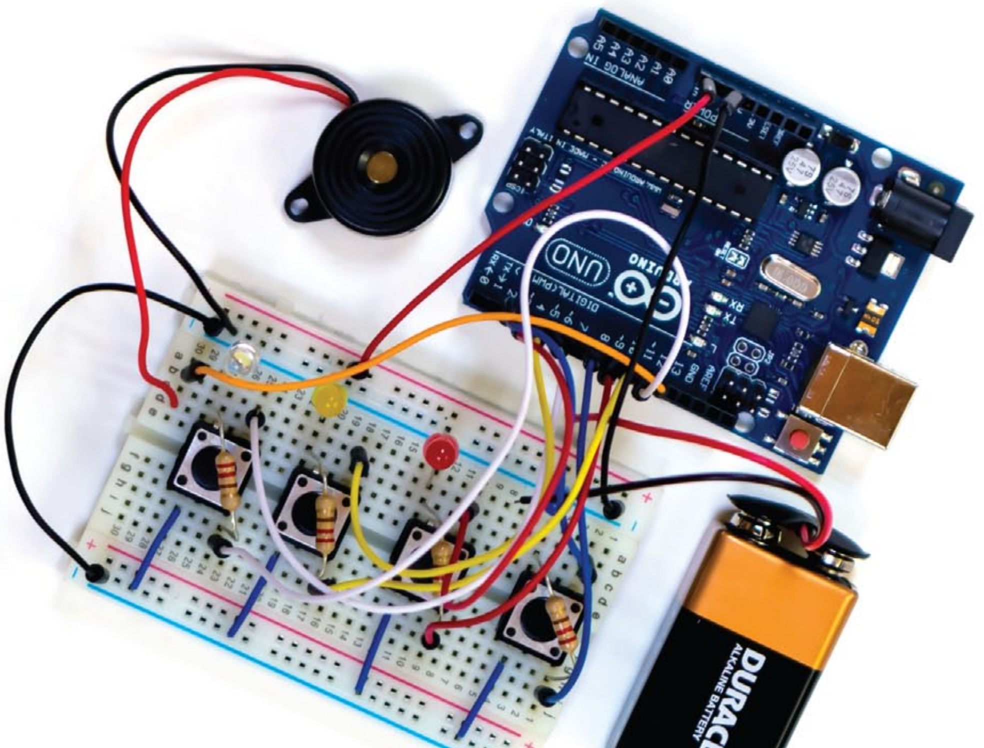
NO STARCH PRESS

THE FINEST IN GEEK ENTERTAINMENT

FEATURING EXCERPTS FROM

ARDUINO PROJECT HANDBOOK

25 PRACTICAL PROJECTS TO GET YOU STARTED

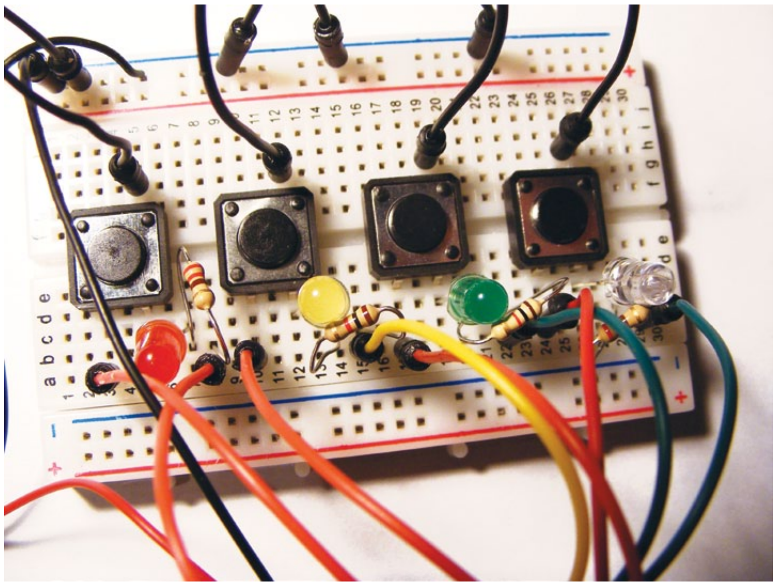


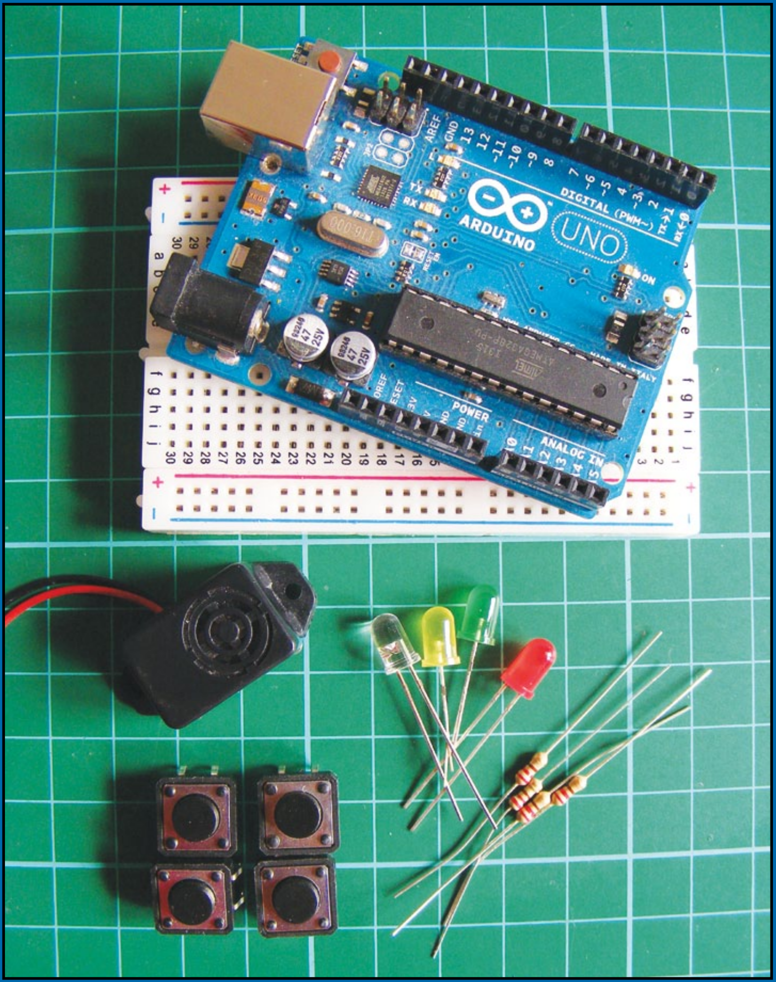
MARK GEDDES

no starch press

PROJECT 8: MEMORY GAME

IN THIS PROJECT WE'LL CREATE OUR OWN VERSION OF AN ATARI ARCADE MEMORY GAME CALLED *TOUCH ME*, USING FOUR LEDS, FOUR PUSHBUTTON SWITCHES, A PIEZO BUZZER, AND SOME RESISTORS AND JUMPER WIRES.





PARTS REQUIRED

- Arduino board
- Breadboard
- Jumper wires
- Piezo buzzer
- 4 momentary tactile four-pin pushbuttons
- 4 LEDs
- 4 220-ohm resistors

LIBRARIES REQUIRED

- Tone

HOW IT WORKS

The original Atari game had four colored panels, each with an LED that lit up in a particular pattern that players had to repeat back (see Figure 8-1).

FIGURE 8-1: The original *Touch Me* game

This memory game plays a short introductory tune and flashes an LED. When you press the correct corresponding button, the lights flash again in a longer sequence. Each time you repeat the sequence back correctly, the game adds an extra step to make the sequence more challenging for you. When you make an error, the game resets itself.

THE BUILD

1. Place the pushbuttons in the breadboard so they straddle the center break with pins A and B on one side of the break, and C and D on the other, as shown in Figure 8-2. (See Project 1 for more information on how the pushbutton works.)

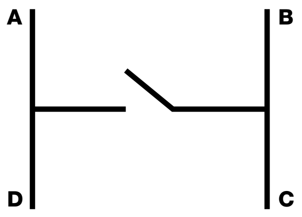


FIGURE 8-2: A pushbutton has four pins.

2. Connect pin B of each pushbutton to the GND rail of your breadboard, and connect the rail to Arduino GND.
3. Connect pin D of each pushbutton to Arduino's digital pins 2 through 5 in order.

4. Insert the LEDs into the breadboard with the shorter, negative legs connected to pin C of each pushbutton. Insert the positive leg into the hole on the right, as shown in the circuit diagram in Figure 12-3.

PUSHBUTTON	ARDUINO/LED
Pin B	GND
Pin C	LED negative legs
Pin D	Arduino pins 2–5

5. Place a 220-ohm resistor into the breadboard with one wire connected to the positive leg of each LED. Connect the other wire of the resistor to the Arduino as follows.

LEDS	ARDUINO/ PUSHBUTTON
Positive legs	Arduino pins 8–11 via 220-ohm resistors
Negative legs	Pushbutton pin C

Make sure the red LED connected to pin 11 is paired with the pushbutton connected to pin 5, the yellow LED connected to pin 10 is paired with the pushbutton connected to pin 4, the green LED connected to pin 9 is paired with the pushbutton connected to pin 3, and the blue LED connected to pin 8 is paired with the pushbutton connected to pin 2.

6. Connect the black wire of the piezo directly to Arduino GND, and the red wire to Arduino pin 12.

PIEZO	ARDUINO
Red wire	Pin 12
Black wire	GND

7. Check your setup against Figure 8-3, and then upload the code in "The Sketch" on page 7.

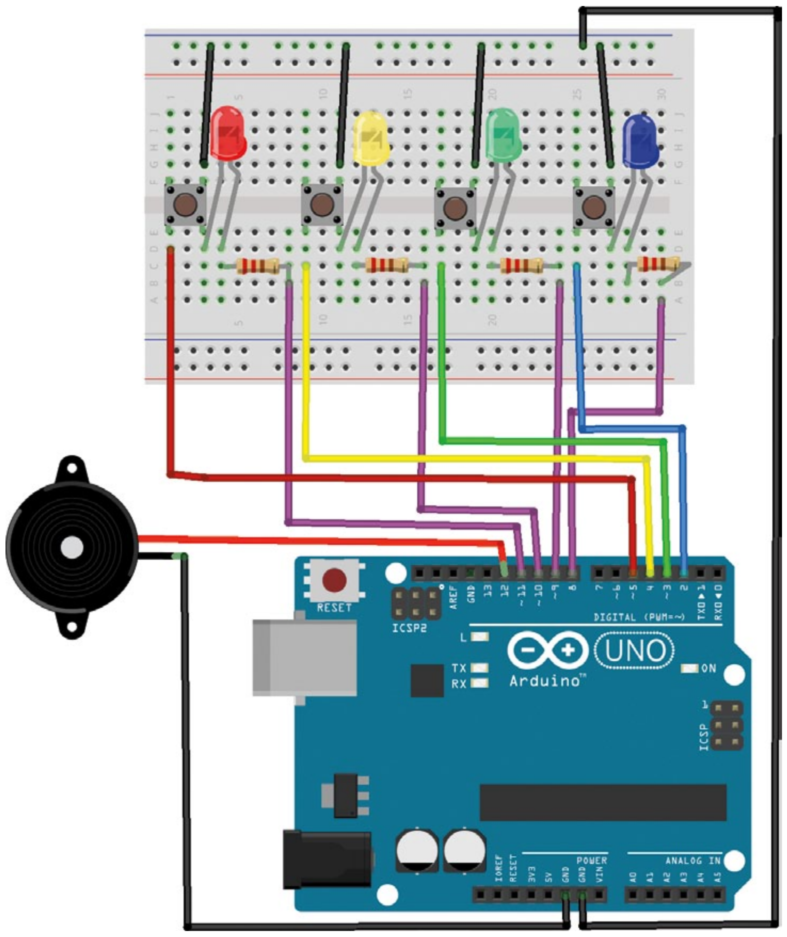


FIGURE 8-3: Circuit diagram for the memory game

THE SKETCH

The sketch generates a random sequence in which the LEDs will light; a random value generated for y in the pattern loop determines which LED is lit (e.g., if y is 2, the LED connected to pin 2 will light). You have to follow and repeat back the pattern to advance to the next level.

In each level, the previous lights are repeated and one more randomly generated light is added to the pattern. Each light is associated with a different tone from the piezo, so you get a different tune each time, too. When you get a sequence wrong, the sketch restarts with a different random sequence. For the sketch to compile correctly, you will need to install the Tone library (available from *http://nostarch.com.com/arduinohandbook/*). See “Libraries” on page 7 for details.

```
--------------------------------------------------------------------
// Used with kind permission from Abdullah Alhazmy www.Alhazmy13.net

#include <Tone.h>
Tone speakerpin;
int starttune[] = {NOTE_C4, NOTE_F4, NOTE_C4, NOTE_F4, NOTE_C4,
                   NOTE_F4, NOTE_C4, NOTE_F4, NOTE_G4, NOTE_F4,
                   NOTE_E4, NOTE_F4, NOTE_G4};
int duration2[] = {100, 200, 100, 200, 100, 400, 100, 100, 100, 100,
                   200, 100, 500};
int note[] = {NOTE_C4, NOTE_C4, NOTE_G4, NOTE_C5, NOTE_G4, NOTE_C5};
int duration[] = {100, 100, 100, 300, 100, 300};
boolean button[] = {2, 3, 4, 5}; // Pins connected to
                                 // pushbutton inputs
boolean ledpin[] = {8, 9, 10, 11}; // Pins connected to LEDs
int turn = 0;          // Turn counter
int buttonstate = 0;   // Check pushbutton state
int randomArray[100]; // Array that can store up to 100 inputs
int inputArray[100];

void setup() {
  Serial.begin(9600);
  speakerpin.begin(12); // Pin connected to piezo buzzer
  for (int x = 0; x < 4; x++) {
    pinMode(ledpin[x], OUTPUT); // Set LED pins as output
  }
  for (int x = 0; x < 4; x++) {
    pinMode(button[x], INPUT); // Set pushbutton pins as inputs
    digitalWrite(button[x], HIGH); // Enable internal pullup;
                                   // pushbuttons start in high
                                   // position; logic reversed
  }
  // Generate "more randomness" with randomArray for the output
  // function so pattern is different each time
  randomSeed(analogRead(0));
  for (int thisNote = 0; thisNote < 13; thisNote ++) {
    speakerpin.play(starttune[thisNote]); // Play the next note
    if (thisNote == 0 || thisNote == 2 || thisNote == 4 ||
        thisNote == 6) { // Hold the note
      digitalWrite(ledpin[0], HIGH);
    }
    if (thisNote == 1 || thisNote == 3 || thisNote == 5 ||
        thisNote == 7 || thisNote == 9 || thisNote == 11) {
      digitalWrite(ledpin[1], HIGH);
    }
    if (thisNote == 8 || thisNote == 12) {
      digitalWrite(ledpin[2], HIGH);
    }
    if (thisNote == 10) {
      digitalWrite(ledpin[3], HIGH);
    }
    delay(duration2[thisNote]);
    speakerpin.stop(); // Stop for the next note
    digitalWrite(ledpin[0], LOW);
    digitalWrite(ledpin[1], LOW);
    digitalWrite(ledpin[2], LOW);
    digitalWrite(ledpin[3], LOW);
    delay(25);
  }
  delay(1000);
}

void loop() {
  // Generate the array to be matched by the player
  for (int y = 0; y <= 99; y++) {
    digitalWrite(ledpin[0], HIGH);
    digitalWrite(ledpin[1], HIGH);
    digitalWrite(ledpin[2], HIGH);
    digitalWrite(ledpin[3], HIGH);
    // Play the next note
    for (int thisNote = 0; thisNote < 6; thisNote ++) {
      speakerpin.play(note[thisNote]); // Hold the note
      delay(duration[thisNote]);       // Stop for the next note
      speakerpin.stop();
      delay(25);
    }
    digitalWrite(ledpin[0], LOW);
    digitalWrite(ledpin[1], LOW);
    digitalWrite(ledpin[2], LOW);
    digitalWrite(ledpin[3], LOW);
    delay(1000);
    // Limited by the turn variable
    for (int y = turn; y <= turn; y++) {
      Serial.println("");
      Serial.print("Turn: ");
      Serial.print(y);
      Serial.println("");
      randomArray[y] = random(1, 5); // Assign a random number (1-4)
      // Light LEDs in random order
      for (int x = 0; x <= turn; x++) {
        Serial.print(randomArray[x]);
        for (int y = 0; y < 4; y++) {
          if (randomArray[x] == 1 && ledpin[y] == 8) {
            digitalWrite(ledpin[y], HIGH);
            speakerpin.play(NOTE_G3, 100);
            delay(400);
            digitalWrite(ledpin[y], LOW);
            delay(100);
          }
          if (randomArray[x] == 2 && ledpin[y] == 9) {
            digitalWrite(ledpin[y], HIGH);
            speakerpin.play(NOTE_A3, 100);
            delay(400);
            digitalWrite(ledpin[y], LOW);
            delay(100);
          }
          if (randomArray[x] == 3 && ledpin[y] == 10) {
            digitalWrite(ledpin[y], HIGH);
```

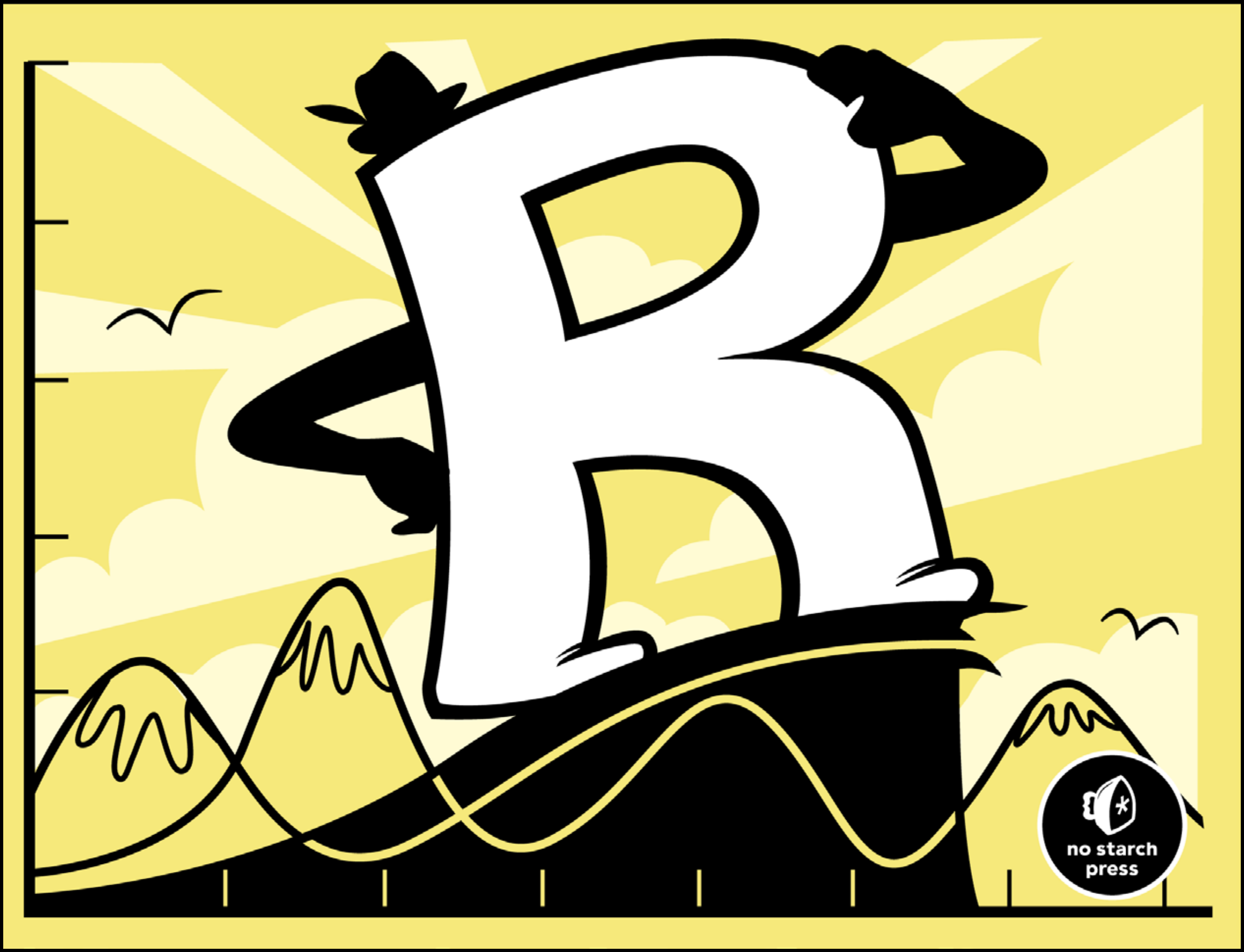
```
              speakerpin.play(NOTE_B3, 100);
              delay(400);
              digitalWrite(ledpin[y], LOW);
              delay(100);
            }
            if (randomArray[x] == 4 && ledpin[y] == 11) {
              digitalWrite(ledpin[y], HIGH);
              speakerpin.play(NOTE_C4, 100);
              delay(400);
              digitalWrite(ledpin[y], LOW);
              delay(100);
            }
          }
        }
      }
      input();
    }
  }

  // Check whether input matches the pattern
  void input() {
    for (int x = 0; x <= turn;) {
      for (int y = 0; y < 4; y++) {
        buttonstate = digitalRead(button[y]); // Check for button push
        if (buttonstate == LOW && button[y] == 2) {
          digitalWrite(ledpin[0], HIGH);
          speakerpin.play(NOTE_G3, 100);
          delay(200);
          digitalWrite(ledpin[0], LOW);
          inputArray[x] = 1;
          delay(250);
          Serial.print(" ");
          Serial.print(1);
          // Check if value of user input matches the generated array
          if (inputArray[x] != randomArray[x]) {
            fail(); // If not, fail function is called
          }
          x++;
        }
        if (buttonstate == LOW && button[y] == 3) {
          digitalWrite(ledpin[1], HIGH);
          speakerpin.play(NOTE_A3, 100);
          delay(200);
          digitalWrite(ledpin[1], LOW);
          inputArray[x] = 2;
          delay(250);
          Serial.print(" ");
          Serial.print(2);
          if (inputArray[x] != randomArray[x]) {
            fail();
          }
          x++;
        }
        if (buttonstate == LOW && button[y] == 4) {
```

```
        digitalWrite(ledpin[2], HIGH);
        speakerpin.play(NOTE_B3, 100);
        delay(200);
        digitalWrite(ledpin[2], LOW);
        inputArray[x] = 3;
        delay(250);
        Serial.print(" ");
        Serial.print(3);
        if (inputArray[x] != randomArray[x]) {
          fail();
        }
        x++;
      }
      if (buttonstate == LOW && button[y] == 5) {
        digitalWrite(ledpin[3], HIGH);
        speakerpin.play(NOTE_C4, 100);
        delay(200);
        digitalWrite(ledpin[3], LOW);
        inputArray[x] = 4;
        delay(250);
        Serial.print(" ");
        Serial.print(4);
        if (inputArray[x] != randomArray[x]) {
          fail();
        }
        x++;
      }
    }
  }
  delay(500);
  turn++; // Increment turn count
}

// Function used if player fails to match the sequence
void fail() {
  for (int y = 0; y <= 2; y++) { // Flash lights to indicate failure
    digitalWrite(ledpin[0], HIGH);
    digitalWrite(ledpin[1], HIGH);
    digitalWrite(ledpin[2], HIGH);
    digitalWrite(ledpin[3], HIGH);
    speakerpin.play(NOTE_G3, 300);
    delay(200);
    digitalWrite(ledpin[0], LOW);
    digitalWrite(ledpin[1], LOW);
    digitalWrite(ledpin[2], LOW);
    digitalWrite(ledpin[3], LOW);
    speakerpin.play(NOTE_C3, 300);
    delay(200);
  }
  delay(500);
  turn = -1; // Reset turn value to start the game again
}
```

THE BOOK OF R

A FIRST COURSE IN PROGRAMMING AND STATISTICS

TILMAN M. DAVIES



2

NUMERICS, ARITHMETIC, ASSIGNMENT, AND VECTORS

In its simplest role, R can function as a mere desktop calculator. In this chapter, I'll discuss how to use the software for arithmetic. I'll also show how to store results so you can use them later in other calculations. Then, you'll learn about vectors, which let you handle multiple values at once. Vectors are an essential tool in R, and much of R's functionality was designed with vector operations in mind. You'll examine some common and useful ways to manipulate vectors and take advantage of vector-oriented behavior.

2.1 R for Basic Math

All common arithmetic operations and mathematical functionality are ready to use at the console prompt. You can perform addition, subtraction, multiplication, and division with the symbols `+`, `-`, `*`, and `/`, respectively. You can create exponents (also referred to as *powers* or *indices*) using `^`, and you control the order of the calculations in a single command using parentheses, `()`.

2.1.1 Arithmetic

In R, standard mathematical rules apply throughout and follow the usual left-to-right order of operations: parentheses, exponents, multiplication, division, addition, subtraction (PEMDAS). Here's an example in the console:

```
R> 2+3
[1] 5
R> 14/6
[1] 2.333333
R> 14/6+5
[1] 7.333333
R> 14/(6+5)
[1] 1.272727
R> 3^2
[1] 9
R> 2^3
[1] 8
```

You can find the square root of any non-negative number with the sqrt function. You simply provide the desired number to x as shown here:

```
R> sqrt(x=9)
[1] 3
R> sqrt(x=5.311)
[1] 2.304561
```

When using R, you'll often find that you need to translate a complicated arithmetic formula into code for evaluation (for example, when replicating a calculation from a textbook or research paper). The next examples provide a mathematically expressed calculation, followed by its execution in R:

$$10^2 + \frac{3 \times 60}{8} - 3$$

```
R> 10^2+3*60/8-3
[1] 119.5
```

$$\frac{5^3 \times (6-2)}{61-3+4}$$

```
R> 5^3*(6-2)/(61-3+4)
[1] 8.064516
```

$$2^{2+1} - 4 + 64^{-2^{2.25-\frac{1}{4}}}$$

```
R> 2^(2+1)-4+64^((-2)^(2.25-1/4))
[1] 16777220
```

$$\left(\frac{0.44 \times (1-0.44)}{34}\right)^{\frac{1}{2}}$$

```
R> (0.44*(1-0.44)/34)^(1/2)
[1] 0.08512966
```

Note that some R expressions require extra parentheses that aren't present in the mathematical expressions. Missing or misplaced parentheses are common causes of arithmetic errors in R, especially when dealing with exponents. If the exponent is itself an arithmetic calculation, it must always appear in parentheses. For example, in the third expression, you need parentheses around `2.25-1/4`. You also need to use parentheses if the number being raised to some power is a calculation, such as the expression 2^{2+1} in the third example. Note that R considers a negative number a calculation because it interprets, for example, `-2` as `-1*2`. This is why you also need the parentheses around `-2` in that same expression. It's important to highlight these issues early because they can easily be overlooked in large chunks of code.

2.1.2 Logarithms and Exponentials

You'll often see or read about researchers performing a *log transformation* on certain data. This refers to rescaling numbers according to the *logarithm*. When supplied a given number x and a value referred to as a *base*, the logarithm calculates the power to which you must raise the base to get to x. For example, the logarithm of $x = 243$ to base 3 (written mathematically as $\log_3 243$) is 5, because $3^5 = 243$. In R, the log transformation is achieved with the `log` function. You supply `log` with the number to transform, assigned to the value `x`, and the base, assigned to `base`, as follows:

```
R> log(x=243,base=3)
[1] 5
```

Here are some things to consider:

- Both x and the base must be positive.
- The log of any number x when the base is equal to x is 1.
- The log of $x = 1$ is always 0, regardless of the base.

There's a particular kind of log transformation often used in mathematics called the *natural log*, which fixes the base at a special mathematical number—*Euler's number*. This is conventionally written as e and is approximately equal to 2.718.

Euler's number gives rise to the *exponential function*, defined as e raised to the power of x, where x can be any number (negative, zero, or positive). The exponential function, $f(x) = e^x$, is often written as $\exp(x)$ and represents the *inverse* of the natural log such that $\exp(\log_e x) = \log_e \exp(x) = x$. The R command for the exponential function is `exp`:

```
R> exp(x=3)
[1] 20.08554
```

The default behavior of `log` is to assume the natural log:

```
R> log(x=20.08554)
[1] 3
```

You must provide the value of `base` yourself if you want to use a value other than e. The logarithm and exponential functions are mentioned here because they become important later on in the book—many statistical methods use them because of their various helpful mathematical properties.

2.1.3 E-Notation

When R prints large or small numbers beyond a certain threshold of significant figures, set at 7 by default, the numbers are displayed using the classic scientific e-notation. The e-notation is typical to most programming languages—and even many desktop calculators—to allow easier interpretation of extreme values. In e-notation, any number x can be expressed as xey, which represents exactly $x \times 10^y$. Consider the number $2,342,151,012,900$. It could, for example, be represented as follows:

- 2.3421510129e12, which is equivalent to writing $2.3421510129 \times 10^{12}$
- 234.21510129e10, which is equivalent to writing $234.21510129 \times 10^{10}$

You could use any value for the power of y, but standard e-notation uses the power that places a decimal just after the first significant digit. Put simply, for a *positive* power $+y$, the e-notation can be interpreted as "move the decimal point y positions to the *right*." For a *negative* power $-y$, the interpretation is "move the decimal point y positions to the *left*." This is exactly how R presents e-notation:

```
R> 2342151012900
[1] 2.342151e+12
R> 0.0000002533
[1] 2.533e-07
```

In the first example, R shows only the first seven significant digits and hides the rest. Note that no information is lost in any calculations even if R hides digits; the e-notation is purely for ease of readability by the user, and the extra digits are still stored by R, even though they aren't shown.

Finally, note that R must impose constraints on how extreme a number can be before it is treated as either infinity (for large numbers) or zero (for small numbers). These constraints depend on your individual system, and I'll discuss the technical details a bit more in Section 6.1.1. However, any modern desktop system can be trusted to be precise enough by default for most computational and statistical endeavors in R.

Exercise 2.1

a. Using R, verify that

$$\frac{6a + 42}{3^{4.2-3.62}} = 29.50556$$

when $a = 2.3$.

b. Which of the following squares negative 4 and adds 2 to the result?
   i. `(-4)^2+2`
   ii. `-4^2+2`
   iii. `(-4)^(2+2)`
   iv. `-4^(2+2)`

c. Using R, how would you calculate the square root of half of the average of the numbers 25.2, 15, 16.44, 15.3, and 18.6?

d. Find $\log_e 0.3$.

e. Compute the exponential transform of your answer to (d).

f. Identify R's representation of -0.00000000423546322 when printing this number to the console.

2.2 Assigning Objects

So far, R has simply displayed the results of the example calculations by printing them to the console. If you want to save the results and perform further operations, you need to be able to *assign* the results of a given computation to an *object* in the current workspace. Put simply, this amounts to storing some item or result under a given name so it can be accessed later, without having to write out that calculation again. In this book, I will use the terms *assign* and *store* interchangeably. Note that some programming books refer to a stored object as a *variable* because of the ability to easily overwrite that object and change it to something different, meaning that what it represents can vary throughout a session. However, I'll use the term *object* throughout this book because we'll discuss variables in Part III as a distinctly different statistical concept.

You can specify an assignment in R in two ways: using arrow notation (`<-`) and using a single equal sign (`=`). Both methods are shown here:

```
R> x <- -5
R> x
[1] -5
```

```
R> x = x + 1  # this overwrites the previous value of x
R> x
[1] -4

R> mynumber = 45.2

R> y <- mynumber*x
R> y
[1] -180.8

R> ls()
[1] "mynumber" "x"        "y"
```

As you can see from these examples, R will display the value assigned to an object when you enter the name of the object into the console. When you use the object in subsequent operations, R will substitute the value you assigned to it. Finally, if you use the ls command (which you saw in Section 1.3.1) to examine the contents of the current workspace, it will reveal the names of the objects in alphabetical order (along with any other previously created items).

Although = and <- do the same thing, it is wise (for the neatness of code if nothing else) to be consistent. Many users choose to stick with the <-, however, because of the potential for confusion in using the = (for example, I clearly didn't mean that x is *mathematically* equal to $x + 1$ earlier). In this book, I'll do the same and reserve = for setting function arguments, which begins in Section 2.3.2. So far you've used only numeric values, but note that the procedure for assignment is universal for all types and classes of objects, which you'll examine in the coming chapters.

Objects can be named almost anything as long as the name begins with a letter (in other words, not a number), avoids symbols (though underscores and periods are fine), and avoids the handful of "reserved" words such as those used for defining special values (see Section 6.1) or for controlling code flow (see Chapter 10). You can find a useful summary of these naming rules in Section 9.1.2.

Exercise 2.2

a. Create an object that stores the value $3^2 \times 4^{1/8}$.
b. Overwrite your object in (a) by itself divided by 2.33. Print the result to the console.
c. Create a new object with the value -8.2×10^{-13}.
d. Print directly to the console the result of multiplying (b) by (c).

2.3 Vectors

Often you'll want to perform the same calculations or comparisons upon multiple entities, for example if you're rescaling measurements in a data set. You could do this type of operation one entry at a time, though this is clearly not ideal, especially if you have a large number of items. R provides a far more efficient solution to this problem with *vectors*.

For the moment, to keep things simple, you'll continue to work with numeric entries only, though many of the utility functions discussed here may also be applied to structures containing non-numeric values. You'll start looking at these other kinds of data in Chapter 4.

2.3.1 Creating a Vector

The vector is the essential building block for handling multiple items in R. In a numeric sense, you can think of a vector as a collection of observations or measurements concerning a single variable, for example, the heights of 50 people or the number of coffees you drink daily. More complicated data structures may consist of several vectors. The function for creating a vector is the single letter `c`, with the desired entries in parentheses separated by commas.

```
R> myvec <- c(1,3,1,42)
R> myvec
[1]  1  3  1 42
```

Vector entries can be calculations or previously stored items (including vectors themselves).

```
R> foo <- 32.1
R> myvec2 <- c(3,-3,2,3.45,1e+03,64^0.5,2+(3-1.1)/9.44,foo)
R> myvec2
[1]    3.000000   -3.000000    2.000000    3.450000 1000.000000    8.000000
[7]    2.201271   32.100000
```

This code created a new vector assigned to the object `myvec2`. Some of the entries are defined as arithmetic expressions, and it's the result of the expression that's stored in the vector. The last element, `foo`, is an existing numeric object defined as `32.1`.

Let's look at another example.

```
R> myvec3 <- c(myvec,myvec2)
R> myvec3
 [1]    1.000000    3.000000    1.000000   42.000000    3.000000   -3.000000
 [7]    2.000000    3.450000 1000.000000    8.000000    2.201271   32.100000
```

This code creates and stores yet another vector, `myvec3`, which contains the entries of `myvec` and `myvec2` appended together in that order.

2.3.2 Sequences, Repetition, Sorting, and Lengths

Here I'll discuss some common and useful functions associated with R vectors: `seq`, `rep`, `sort`, and `length`.

Let's create an equally spaced sequence of increasing or decreasing numeric values. This is something you'll need often, for example when programming loops (see Chapter 10) or when plotting data points (see Chapter 7). The easiest way to create such a sequence, with numeric values separated by intervals of 1, is to use the colon operator.

```
R> 3:27
 [1]  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
```

The example `3:27` should be read as "from 3 to 27 (by 1)." The result is a numeric vector just as if you had listed each number manually in parentheses with `c`. As always, you can also provide either a previously stored value or a (strictly parenthesized) calculation when using the colon operator:

```
R> foo <- 5.3
R> bar <- foo:(-47+1.5)
R> bar
 [1]   5.3   4.3   3.3   2.3   1.3   0.3  -0.7  -1.7  -2.7  -3.7  -4.7
[12]  -5.7  -6.7  -7.7  -8.7  -9.7 -10.7 -11.7 -12.7 -13.7 -14.7 -15.7
[23] -16.7 -17.7 -18.7 -19.7 -20.7 -21.7 -22.7 -23.7 -24.7 -25.7 -26.7
[34] -27.7 -28.7 -29.7 -30.7 -31.7 -32.7 -33.7 -34.7 -35.7 -36.7 -37.7
[45] -38.7 -39.7 -40.7 -41.7 -42.7 -43.7 -44.7
```

Sequences with seq

You can also use the `seq` command, which allows for more flexible creations of sequences. This ready-to-use function takes in a `from` value, a `to` value, and a `by` value, and it returns the corresponding sequence as a numeric vector.

```
R> seq(from=3,to=27,by=3)
[1]  3  6  9 12 15 18 21 24 27
```

This gives you a sequence with intervals of 3 rather than 1. Note that these kinds of sequences will always start at the `from` number but will not always include the `to` number, depending on what you are asking R to increase (or decrease) them `by`. For example, if you are increasing (or decreasing) by even numbers and your sequence ends in an odd number, the final number won't be included. Instead of providing a `by` value, however, you can specify a `length.out` value to produce a vector with that many numbers, evenly spaced between the `from` and `to` values.

```
R> seq(from=3,to=27,length.out=40)
 [1]  3.000000  3.615385  4.230769  4.846154  5.461538  6.076923  6.692308
 [8]  7.307692  7.923077  8.538462  9.153846  9.769231 10.384615 11.000000
[15] 11.615385 12.230769 12.846154 13.461538 14.076923 14.692308 15.307692
```

```
[22] 15.923077 16.538462 17.153846 17.769231 18.384615 19.000000 19.615385
[29] 20.230769 20.846154 21.461538 22.076923 22.692308 23.307692 23.923077
[36] 24.538462 25.153846 25.769231 26.384615 27.000000
```

By setting `length.out` to `40`, you make the program print exactly 40 evenly spaced numbers from 3 to 27.

For decreasing sequences, the use of `by` must be negative. Here's an example:

```
R> foo <- 5.3
R> myseq <- seq(from=foo,to=(-47+1.5),by=-2.4)
R> myseq
 [1]   5.3   2.9   0.5  -1.9  -4.3  -6.7  -9.1 -11.5 -13.9 -16.3 -18.7 -21.1
[13] -23.5 -25.9 -28.3 -30.7 -33.1 -35.5 -37.9 -40.3 -42.7 -45.1
```

This code uses the previously stored object `foo` as the value for `from` and uses the parenthesized calculation `(-47+1.5)` as the `to` value. Given those values (that is, with `foo` being greater than `(-47+1.5)`), the sequence can progress only in negative steps; directly above, we set `by` to be `-2.4`. The use of `length.out` to create decreasing sequences, however, remains the same (it would make no sense to specify a "negative length"). For the same `from` and `to` values, you can create a decreasing sequence of length 5 easily, as shown here:

```
R> myseq2 <- seq(from=foo,to=(-47+1.5),length.out=5)
R> myseq2
[1]   5.3  -7.4 -20.1 -32.8 -45.5
```

There are shorthand ways of calling these functions, which you'll learn about in Chapter 9, but in these early stages I'll stick with the explicit usage.

Repetition with rep

Sequences are extremely useful, but sometimes you may want simply to repeat a certain value. You do this using `rep`.

```
R> rep(x=1,times=4)
[1] 1 1 1 1
R> rep(x=c(3,62,8.3),times=3)
[1]  3.0 62.0  8.3  3.0 62.0  8.3  3.0 62.0  8.3
R> rep(x=c(3,62,8.3),each=2)
[1]  3.0  3.0 62.0 62.0  8.3  8.3
R> rep(x=c(3,62,8.3),times=3,each=2)
 [1]  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0
[16] 62.0  8.3  8.3
```

The `rep` function is given a single value or a vector of values as its argument `x`, as well as a value for the arguments `times` and `each`. The value for `times` provides the number of times to repeat `x`, and `each` provides the number of times to repeat each element of x. In the first line directly above, you simply repeat a single value four times. The other examples first use rep and times on a vector to repeat the entire vector, then use each to repeat each member of the vector, and finally use both times and each to do both at once.

If neither times nor each is specified, R's default is to treat the values of times and each as 1 so that a call of rep(x=c(3,62,8.3)) will just return the originally supplied x with no changes.

As with seq, you can include the result of rep in a vector of the same data type, as shown in the following example:

```
R> foo <- 4
R> c(3,8.3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1))
[1]  3.00  8.30 32.00 32.00 32.00 32.00 -2.00 -1.25 -0.50  0.25  1.00
```

Here, I've constructed a vector where the third to sixth entries (inclusive) are governed by the evaluation of a rep command—the single value 32 repeated foo times (where foo is stored as 4). The last five entries are the result of an evaluation of seq, namely a sequence from −2 to 1 of length foo+1 (5).

Sorting with sort

Sorting a vector in increasing or decreasing order of its elements is another simple operation that crops up in everyday tasks. The conveniently named sort function does just that.

```
R> sort(x=c(2.5,-1,-10,3.44),decreasing=FALSE)
[1] -10.00  -1.00   2.50   3.44

R> sort(x=c(2.5,-1,-10,3.44),decreasing=TRUE)
[1]   3.44   2.50  -1.00 -10.00

R> foo <- seq(from=4.3,to=5.5,length.out=8)
R> foo
[1] 4.300000 4.471429 4.642857 4.814286 4.985714 5.157143 5.328571 5.500000
R> bar <- sort(x=foo,decreasing=TRUE)
R> bar
[1] 5.500000 5.328571 5.157143 4.985714 4.814286 4.642857 4.471429 4.300000

R> sort(x=c(foo,bar),decreasing=FALSE)
 [1] 4.300000 4.300000 4.471429 4.471429 4.642857 4.642857 4.814286 4.814286
 [9] 4.985714 4.985714 5.157143 5.157143 5.328571 5.328571 5.500000 5.500000
```

The sort function is pretty straightforward. You supply a vector to the function as the argument x, and a second argument, decreasing, indicates the order in which you want to sort. This argument takes a type of value you have not yet met: one of the all-important *logical* values. A logical value can be only one of two specific, case-sensitive values: TRUE or FALSE. Generally speaking, logicals are used to indicate the satisfaction or failure of a certain *condition*, and they form an integral part of all programming languages. You'll investigate logical values in R in greater detail in Section 4.1. For now, in regards to sort, you set decreasing=FALSE to sort from smallest to largest, and decreasing=TRUE sorts from largest to smallest.

Finding a Vector Length with length

I'll round off this section with the length function, which determines how many entries exist in a vector given as the argument x.

```
R> length(x=c(3,2,8,1))
[1] 4

R> length(x=5:13)
[1] 9

R> foo <- 4
R> bar <- c(3,8.3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1))
R> length(x=bar)
[1] 11
```

Note that if you include entries that depend on the evaluation of other functions (in this case, calls to rep and seq), length tells you the number of entries *after* those inner functions have been executed.

Exercise 2.3

a. Create and store a sequence of values from 5 to −11 that progresses in steps of 0.3.
b. Overwrite the object from (a) using the same sequence with the order reversed.
c. Repeat the vector c(-1,3,-5,7,-9) twice, with each element repeated 10 times, and store the result. Display the result sorted from largest to smallest.
d. Create and store a vector that contains, in any configuration, the following:
   i. A sequence of integers from 6 to 12 (inclusive)
   ii. A threefold repetition of the value 5.3
   iii. The number −3
   iv. A sequence of nine values starting at 102 and ending at the number that is the total length of the vector created in (c)
e. Confirm that the length of the vector created in (d) is 20.

2.3.3 Subsetting and Element Extraction

In all the results you have seen printed to the console screen so far, you may have noticed a curious feature. Immediately to the left of the output there is a square-bracketed [1]. When the output is a long vector that spans the width of the console and wraps onto the following line, another square-bracketed number appears to the left of the new line. These numbers represent the *index* of the entry directly to the right. Quite simply, the index corresponds to the *position* of a value within a vector, and that's precisely why the first value always has a [1] next to it (even if it's the only value and not part of a larger vector).

These indexes allow you to retrieve specific elements from a vector, which is known as *subsetting*. Suppose you have a vector called myvec in your workspace. Then there will be exactly length(x=myvec) entries in myvec, with each entry having a specific position: 1 or 2 or 3, all the way up to length(x=myvec). You can access individual elements by asking R to return the values of myvec at specific locations, done by entering the name of the vector followed by the position in square brackets.

```
R> myvec <- c(5,-2.3,4,4,4,6,8,10,40221,-8)
R> length(x=myvec)
[1] 10
R> myvec[1]
[1] 5

R> foo <- myvec[2]
R> foo
[1] -2.3

R> myvec[length(x=myvec)]
[1] -8
```

Because length(x=myvec) results in the final index of the vector (in this case, 10), entering this phrase in the square brackets extracts the final element, -8. Similarly, you could extract the second-to-last element by subtracting 1 from the length; let's try that, and also assign the result to a new object:

```
R> myvec.len <- length(x=myvec)
R> bar <- myvec[myvec.len-1]
R> bar
[1] 40221
```

As these examples show, the index may be an arithmetic function of other numbers or previously stored values. You can assign the result to a new object in your workspace in the usual way with the <- notation. Using your knowledge of sequences, you can use the colon notation with the length of the specific vector to obtain all possible indexes for extracting a particular element in the vector:

```
R> 1:myvec.len
[1]  1  2  3  4  5  6  7  8  9 10
```

You can also delete individual elements by using *negative* versions of the indexes supplied in the square brackets. Continuing with the objects `myvec`, `foo`, `bar`, and `myvec.len` as defined earlier, consider the following operations:

```
R> myvec[-1]
[1]    -2.3     4.0     4.0     4.0     6.0     8.0    10.0 40221.0    -8.0
```

This line produces the contents of `myvec` without the first element. Similarly, the following code assigns to the object `baz` the contents of `myvec` without its second element:

```
R> baz <- myvec[-2]
R> baz
[1]     5     4     4     4     6     8    10 40221    -8
```

Again, the index in the square brackets can be the result of an appropriate calculation, like so:

```
R> qux <- myvec[-(myvec.len-1)]
R> qux
[1]  5.0 -2.3  4.0  4.0  4.0  6.0  8.0 10.0 -8.0
```

Using the square-bracket operator to extract or delete values from a vector does not change the original vector you are subsetting *unless* you explicitly overwrite the vector with the subsetted version. For instance, in this example, `qux` is a new vector defined as `myvec` without its second-to-last entry, but in your workspace, `myvec` itself *remains unchanged*. In other words, subsetting vectors in this way simply returns the requested elements, which can be assigned to a new object if you want, but doesn't alter the original object in the workspace.

Now, suppose you want to piece `myvec` back together from `qux` and `bar`. You can call something like this:

```
R> c(qux[-length(x=qux)],bar,qux[length(x=qux)])
 [1]     5.0    -2.3     4.0     4.0     4.0     6.0     8.0    10.0 40221.0
[10]    -8.0
```

As you can see, this line uses `c` to reconstruct the vector in three parts: `qux[-length(x=qux)]`, the object `bar` defined earlier, and `qux[length(x=qux)]`. For clarity, let's examine each part in turn.

- `qux[-length(x=qux)]`

  This piece of code returns the values of `qux` except for its last element.

  ```
  R> length(x=qux)
  [1] 9
  R> qux[-length(x=qux)]
  [1]  5.0 -2.3  4.0  4.0  4.0  6.0  8.0 10.0
  ```

  Now you have a vector that's the same as the first eight entries of `myvec`.

- `bar`

  Earlier, you had stored `bar` as the following:

  ```
  R> bar <- myvec[myvec.len-1]
  R> bar
  [1] 40221
  ```

  This is precisely the second-to-last element of `myvec` that `qux` is missing. So, you'll slot this value in after `qux[-length(x=qux)]`.

- `qux[length(x=qux)]`

  Finally, you just need the last element of `qux` that matches the last element of `myvec`. This is extracted from `qux` (not deleted as earlier) using `length`.

  ```
  R> qux[length(x=qux)]
  [1] -8
  ```

Now it should be clear how calling these three parts of code together, in this order, is one way to reconstruct `myvec`.

As with most operations in R, you are not restricted to doing things one by one. You can also subset objects using *vectors of indexes*, rather than individual indexes. Using `myvec` again from earlier, you get the following:

```
R> myvec[c(1,3,5)]
[1] 5 4 4
```

This returns the first, third, and fifth elements of `myvec` in one go. Another common and convenient subsetting tool is the colon operator (discussed in Section 2.3.2), which creates a sequence of indexes. Here's an example:

```
R> 1:4
[1] 1 2 3 4
R> foo <- myvec[1:4]
R> foo
[1]  5.0 -2.3  4.0  4.0
```

This provides the first four elements of `myvec` (recall that the colon operator returns a numeric vector, so there is no need to explicitly wrap this using `c`).

The order of the returned elements depends entirely upon the index vector supplied in the square brackets. For example, using `foo` again, consider the order of the indexes and the resulting extractions, shown here:

```
R> length(x=foo):2
[1] 4 3 2
R> foo[length(foo):2]
[1]  4.0  4.0 -2.3
```

Here you extracted elements starting at the end of the vector, working backward. You can also use `rep` to repeat an index, as shown here:

```
R> indexes <- c(4,rep(x=2,times=3),1,1,2,3:1)
R> indexes
 [1] 4 2 2 2 1 1 2 3 2 1
R> foo[indexes]
 [1]  4.0 -2.3 -2.3 -2.3  5.0  5.0 -2.3  4.0 -2.3  5.0
```

This is now something a little more general than strictly "subsetting"—by using an index vector, you can create an entirely new vector of any length consisting of some or all of the elements in the original vector. As shown earlier, this index vector can contain the desired element positions in any order and can repeat indexes.

You can also return the elements of a vector after deleting more than one element. For example, to create a vector after removing the first and third elements of `foo`, you can execute the following:

```
R> foo[-c(1,3)]
[1] -2.3  4.0
```

Note that it is not possible to mix positive and negative indexes in a single index vector.

Sometimes you'll need to overwrite certain elements in an existing vector with new values. In this situation, you first specify the elements you want to overwrite using square brackets and then use the assignment operator to assign the new values. Here's an example:

```
R> bar <- c(3,2,4,4,1,2,4,1,0,0,5)
R> bar
 [1] 3 2 4 4 1 2 4 1 0 0 5
R> bar[1] <- 6
R> bar
 [1] 6 2 4 4 1 2 4 1 0 0 5
```

This overwrites the first element of bar, which was originally 3, with a new value, 6. When selecting multiple elements, you can specify a single value to replace them all or enter a vector of values that's equal in length to the number of elements selected to replace them one for one. Let's try this with the same bar vector from earlier.

```
R> bar[c(2,4,6)] <- c(-2,-0.5,-1)
R> bar
 [1]  6.0 -2.0  4.0 -0.5  1.0 -1.0  4.0  1.0  0.0  0.0  5.0
```

Here you overwrite the second, fourth, and sixth elements with -2, -0.5, and -1, respectively; all else remains the same. By contrast, the following code overwrites elements 7 to 10 (inclusive), replacing them all with 100:

```
R> bar[7:10] <- 100
R> bar
 [1]   6.0  -2.0   4.0  -0.5   1.0  -1.0 100.0 100.0 100.0 100.0   5.0
```

Finally, it's important to mention that this section has focused on just one of the two main methods, or "flavors," of vector element extraction in R. You'll look at the alternative method, using logical flags, in Section 4.1.5.

Exercise 2.4

a. Create and store a vector that contains the following, in this order:
   – A sequence of length 5 from 3 to 6 (inclusive)
   – A twofold repetition of the vector c(2,-5.1,-33)
   – The value $\frac{7}{42} + 2$
b. Extract the first and last elements of your vector from (a), storing them as a new object.
c. Store as a third object the values returned by omitting the first and last values of your vector from (a).
d. Use only (b) and (c) to reconstruct (a).
e. Overwrite (a) with the same values sorted from smallest to largest.
f. Use the colon operator as an index vector to reverse the order of (e), and confirm this is identical to using sort on (e) with decreasing=TRUE.
g. Create a vector from (c) that repeats the third element of (c) three times, the sixth element four times, and the last element once.

h. Create a new vector as a copy of (e) by assigning (e) as is to a newly named object. Using this new copy of (e), overwrite the first, the fifth to the seventh (inclusive), and the last element with the values 99 to 95 (inclusive), respectively.

2.3.4 Vector-Oriented Behavior

Vectors are so useful because they allow R to carry out operations on multiple elements simultaneously with speed and efficiency. This *vector-oriented*, *vectorized*, or *element-wise* behavior is a key feature of the language, one that you will briefly examine here through some examples of rescaling measurements.

Let's start with this simple example:

```
R> foo <- 5.5:0.5
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
R> foo-c(2,4,6,8,10,12)
[1]   3.5   0.5  -2.5  -5.5  -8.5 -11.5
```

This code creates a sequence of six values between 5.5 and 0.5, in increments of 1. From this vector, you subtract another vector containing 2, 4, 6, 8, 10, and 12. What does this do? Well, quite simply, R matches up the elements according to their respective positions and performs the operation on each corresponding pair of elements. The resulting vector is obtained by subtracting the first element of `c(2,4,6,8,10,12)` from the first element of `foo` (5.5 − 2 = 3.5), then by subtracting the second element of `c(2,4,6,8,10,12)` from the second element of `foo` (4.5 − 4 = 0.5), and so on. Thus, rather than inelegantly cycling through each element in turn (as you could do by hand or by explicitly using a loop), R permits a fast and efficient alternative using vector-oriented behavior. Figure 2-1 illustrates how you can understand this type of calculation and highlights the fact that the positions of the elements are crucial in terms of the final result; elements in differing positions have no effect on one another.

The situation is made more complicated when using vectors of different lengths, which can happen in two distinct ways. The first is when the length of the longer vector can be evenly divided by the length of the shorter vector. The second is when the length of the longer vector *cannot* be divided by the length of the shorter vector—this is usually unintentional on the user's part. In both of these situations, R essentially attempts to replicate, or *recycle*, the shorter vector by as many times as needed to match the length of the longer vector, before completing the specified operation. As an example, suppose you wanted to alternate the entries of `foo` shown earlier as negative

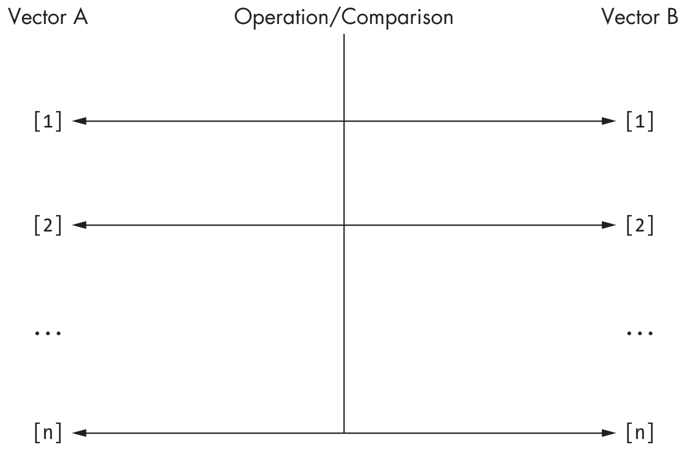


Figure 2-1: A conceptual diagram of the element-wise behavior of a comparison or operation carried out on two vectors of equal length in R. Note that the operation is performed by matching up the element positions.

and positive. You could explicitly multiply `foo` by `c(1,-1,1,-1,1,-1)`, but you don't need to write out the full latter vector. Instead, you can write the following:

```
R> bar <- c(1,-1)
R> foo*bar
[1]  5.5 -4.5  3.5 -2.5  1.5 -0.5
```

Here `bar` has been applied repeatedly throughout the length of `foo` until completion. The left plot of Figure 2-2 illustrates this particular example. Now let's see what happens when the vector lengths are not evenly divisible.

```
R> baz <- c(1,-1,0.5,-0.5)
R> foo*baz
[1]  5.50 -4.50  1.75 -1.25  1.50 -0.50
Warning message:
In foo * baz :
  longer object length is not a multiple of shorter object length
```

Here you see that R has matched the first four elements of `foo` with the entirety of `baz`, but it's not able to fully repeat the vector again. The repetition has been attempted, with the first two elements of `baz` being matched with the last two of the longer `foo`, though not without a protest from R, which notifies the user of the unevenly divisible lengths (you'll look at warnings in more detail in Section 12.1). The plot on the right in Figure 2-2 illustrates this example.

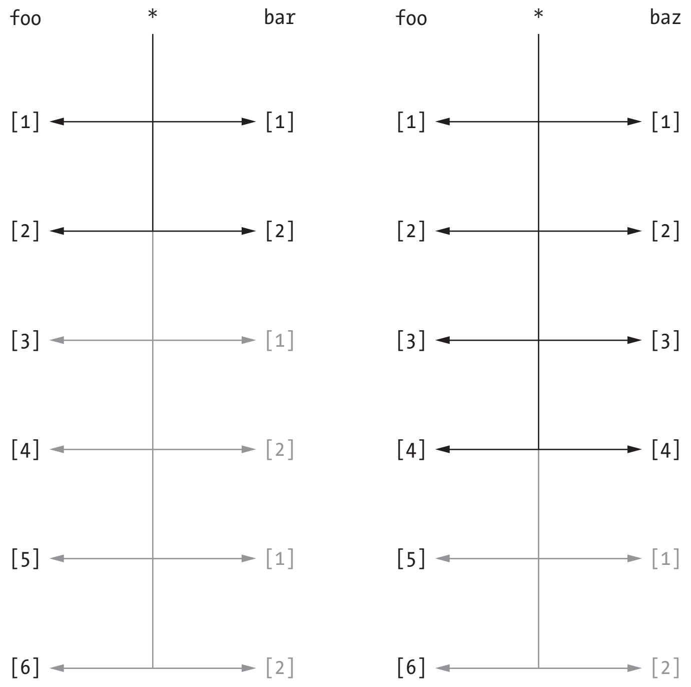


Figure 2-2: An element-wise operation on two vectors of differing lengths. Left: `foo` *multiplied by* `bar`*; lengths are evenly divisible. Right:* `foo` *multiplied by* `baz`*; lengths are not evenly divisible, and a warning is issued.*

As I noted in Section 2.3.3, you can consider single values to be vectors of length 1, so you can use a single value to repeat an operation on all the values of a vector of any length. Here's an example, using the same vector `foo`:

```
R> qux <- 3
R> foo+qux
[1] 8.5 7.5 6.5 5.5 4.5 3.5
```

This is far easier than executing `foo+c(3,3,3,3,3,3)` or the more general `foo+rep(x=3,times=length(x=foo))`. Operating on vectors using a single value in this fashion is quite common, such as if you want to rescale or translate a set of measurements by some constant amount.

Another benefit of vector-oriented behavior is that you can use vectorized functions to complete potentially laborious tasks. For example, if you want to sum or multiply all the entries in a numeric vector, you can just use a built-in function.

Recall foo, shown earlier:

```
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
```

You can find the sum of these six elements with

```
R> sum(foo)
[1] 18
```

and their product with

```
R> prod(foo)
[1] 162.4219
```

Far from being just convenient, vectorized functions are faster and more efficient than an explicitly coded iterative approach like a loop. The main takeaway from these examples is that much of R's functionality is designed specifically for certain data structures, ensuring neatness of code as well as optimization of performance.

Lastly, as mentioned earlier, this vector-oriented behavior applies in the same way to overwriting multiple elements. Again using foo, examine the following:

```
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
R> foo[c(1,3,5,6)] <- c(-99,99)
R> foo
[1] -99.0   4.5  99.0   2.5 -99.0  99.0
```

You see four specific elements being overwritten by a vector of length 2, which is recycled in the same fashion you're familiar with. Again, the length of the vector of replacements must evenly divide the number of elements being overwritten, or else a warning similar to the one shown earlier will be issued when R cannot complete a full-length recycle.

Exercise 2.5

a. Convert the vector c(2,0.5,1,2,0.5,1,2,0.5,1) to a vector of only 1s, using a vector of length 3.

b. The conversion from a temperature measurement in degrees Fahrenheit F to Celsius C is performed using the following equation:

$$C = \frac{5}{9}(F - 32)$$

Use vector-oriented behavior in R to convert the temperatures 45, 77, 20, 19, 101, 120, and 212 in degrees Fahrenheit to degrees Celsius.

c. Use the vector `c(2,4,6)` and the vector `c(1,2)` in conjunction with `rep` and `*` to produce the vector `c(2,4,6,4,8,12)`.

d. Overwrite the middle four elements of the resulting vector from (c) with the two recycled values `-0.1` and `-100`, in that order.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`+, *, -, /, ^`	Arithmetic	Section 2.1, p. 17
`sqrt`	Square root	Section 2.1.1, p. 18
`log`	Logarithm	Section 2.1.2, p. 19
`exp`	Exponential	Section 2.1.2, p. 19
`<-, =`	Object assignment	Section 2.2, p. 21
`c`	Vector creation	Section 2.3.1, p. 23
`:, seq`	Sequence creation	Section 2.3.2, p. 24
`rep`	Value/vector repetition	Section 2.3.2, p. 25
`sort`	Vector sorting	Section 2.3.2, p. 26
`length`	Determine vector length	Section 2.3.2, p. 27
`[ ]`	Vector subsetting/extraction	Section 2.3.3, p. 28
`sum`	Sum all vector elements	Section 2.3.4, p. 36
`prod`	Multiply all vector elements	Section 2.3.4, p. 36

CODING IPHONE APPS FOR KIDS

A PLAYFUL INTRODUCTION TO SWIFT

GLORIA WINQUIST AND MATT MCCARTHY

no starch press

2
LEARNING TO CODE IN A PLAYGROUND

A "Hello World" app is no small accomplishment, but now it's time to really learn how to write some code. Xcode provides a special type of document called a *playground*, which is a great place to learn how to program using Swift. In a playground, you can write lines of code and immediately see what happens when that code runs, without going through the trouble of writing a whole app, as we did in Chapter 1.

Let's get started by opening up a playground. Open Xcode and select **Get started with a playground**, as shown in the Welcome to Xcode dialog in Figure 2-1. If this window doesn't automatically open for you when you launch Xcode, select **Welcome to Xcode** from the Window option in the menu or press ⌘-SHIFT-1.

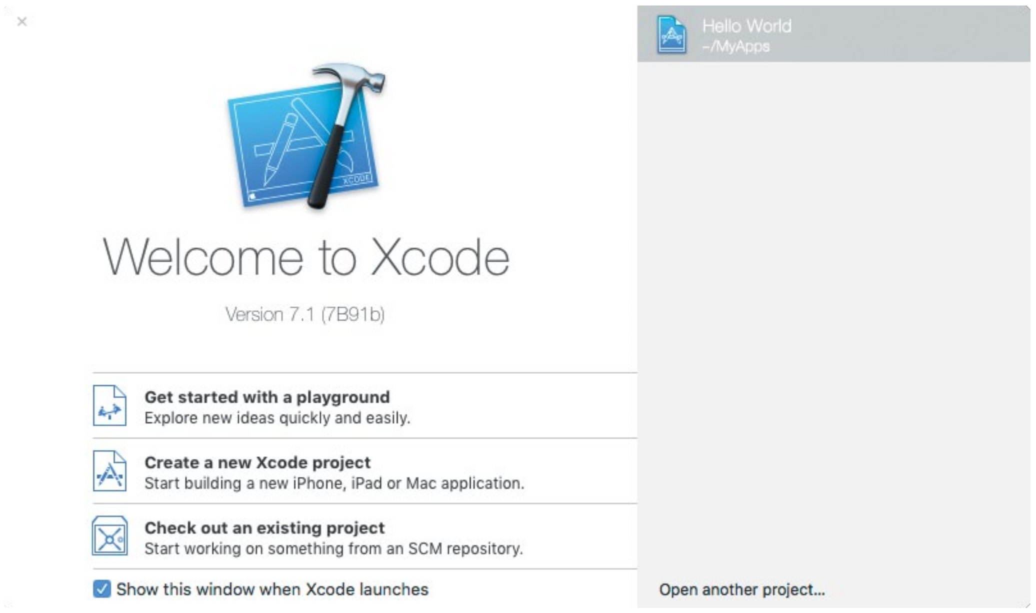
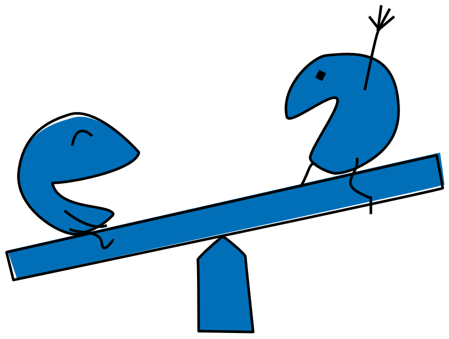
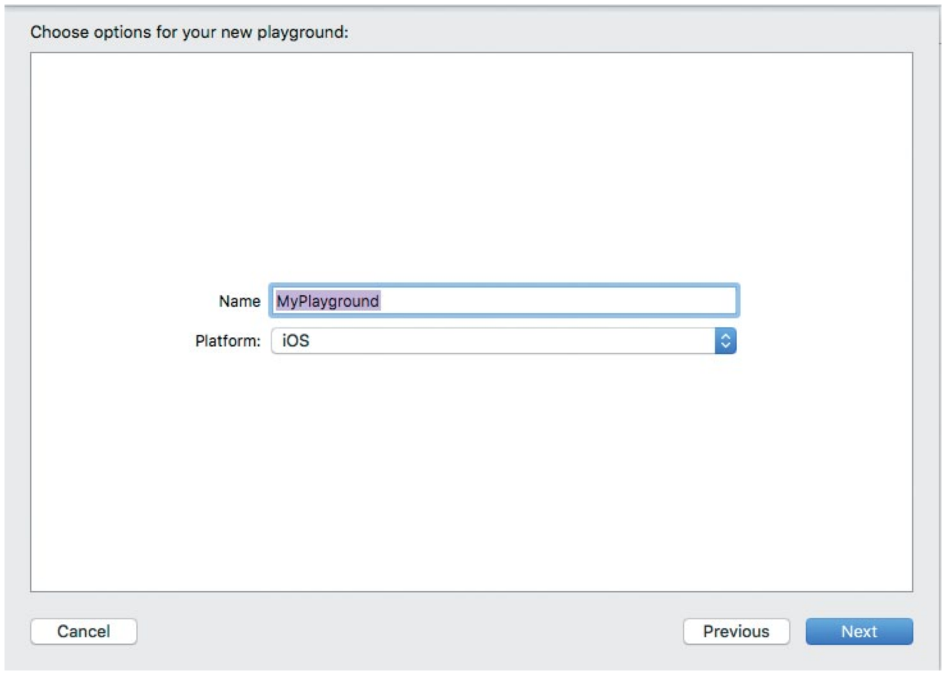

Hello World
~/MyApps
Welcome to Xcode
Version 7.1 (7B91b)
Get started with a playground
Explore new ideas quickly and easily.
Create a new Xcode project
Start building a new iPhone, iPad or Mac application.
Check out an existing project
Start working on something from an SCM repository.
Show this window when Xcode launches
Open another project...

Figure 2-1: Getting started with a playground

You'll be asked to name your playground (Figure 2-2). In this example, we'll keep the name *MyPlayground*, but feel free to name it whatever you want. Make sure that you choose iOS as the platform to run the playground.

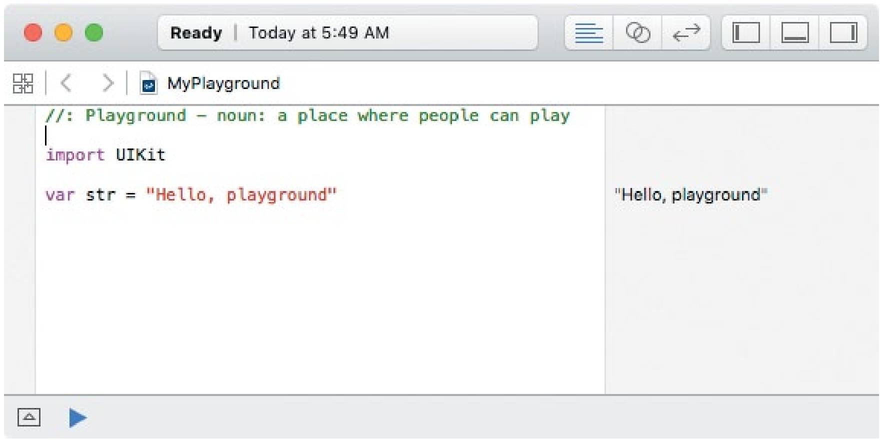




Name: MyPlayground
Platform: iOS

Figure 2-2: Naming the playground and selecting the platform

When the playground first opens, you'll see two panels in the window, just like in Figure 2-3. On the left is the playground editor, where you'll write your code. On the right is the results sidebar, which displays the results of your code.



```
//: Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"
```

```
"Hello, playground"
```

Figure 2-3: Playground editor and results sidebar

The line `var str = "Hello, playground"` in Figure 2-3 creates a variable named `str`. A *variable* is like a container; you can use it to hold almost anything—a simple number, a string of letters, or a complex object. Let's take a closer look at how variables work.

CONSTANTS AND VARIABLES

Here's the line of code from Figure 2-3 again:

```
var str = "Hello, playground"          "Hello, playground"
```

It does two things. First, it creates a variable named `str`. This is called a *declaration* because we are declaring that we would like to create a variable. To create a variable, you type the word `var` and then type a name for your variable. In this case, we named it `str`. There are some rules when it comes to naming variables, but we'll go over them later, so for now stick with this example.

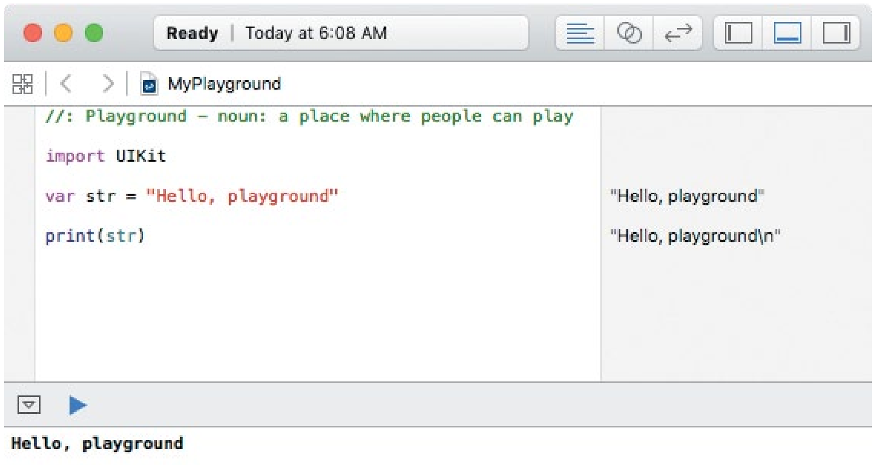
Second, this line of code gives a value of `"Hello, playground"` to `str` using the `=` operator. This is called an *assignment* because we are assigning a value to our newly created variable. Remember, you can think of a variable as a container that holds something. So now we have a container named `str` that holds `"Hello, playground"`.

You can read this line of code as "the variable `str` equals `Hello, playground`." As you can see, Swift is often very readable; this line of code practically tells you in English what it's doing.

Variables are handy because now if you want to print the words "Hello, playground" all you have to do is use the command `print` on `str`, just like in the following code:

```
print(str)                        "Hello, playground\n"
```

This prints `"Hello, playground\n"` in the results sidebar. The `\n` is added automatically to the end of whatever you print. It is known as the *newline* character and tells the computer to go to a new line. To see the results of your program as it would actually run, bring up the debug area, which will appear below the two panels as shown in Figure 2-4. To do this, go to **View ▸ Debug Area ▸ Show Debug Area** in the Xcode menu or press ⌘-SHIFT-Y. When `str` is printed in the debug area, you can see that the quotes around `Hello, playground` and the newline character do not appear. This is what `str` would really look like if you were to officially compile and run this program!



```
//: Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"                "Hello, playground"

print(str)                                   "Hello, playground\n"
```

```
Hello, playground
```

Figure 2-4: Viewing the real output of your program in the debug area

Variables can change (or *vary*!) in your programs, so you can change the value of a variable if you want it to hold something else. Let's try that now. Add these lines to your playground program:

```
❶ str = "Hello, world"        "Hello, world"
  print(str)                  "Hello, world\n"
```

To change the value of a variable, type its name and use the `=` operator to set it to a new value. We do this at ❶ to change the value of `str` to `"Hello, world"`. The computer throws away whatever `str` used to hold, and says, "Okay, boss, `str` is now `Hello, world`" (that is, it would say that if it could talk!).

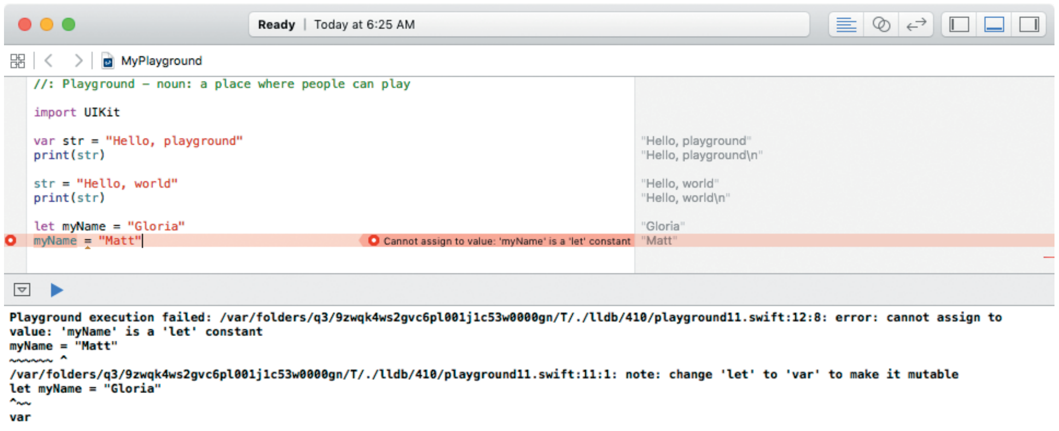
Notice that when we change the value of `str`, we don't write `var` again. The computer remembers that we declared `str` in a previous line of code and knows that `str` already exists. So we don't need to create `str` again, we just want to put something different in it.

You can also declare *constants*. Like variables, constants hold values. The big difference between a constant and a variable is that a constant can never change its value. Variables can vary, and constants are, well, constant! Declaring a constant is similar to declaring a variable, but we use the word `let` instead of `var`:

`let myName = "Gloria"`	`"Gloria"`

Here we create a constant called `myName` and assign it the value of `"Gloria"`.

Once you create a constant and give it a value, it will have that value until the end of time. Think of a constant as a big rock into which you've carved your value. If you try to give `myName` another value, like `"Matt"`, you'll get an error like the one in Figure 2-5.



```
//: Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"          "Hello, playground"
print(str)                             "Hello, playground\n"

str = "Hello, world"                   "Hello, world"
print(str)                             "Hello, world\n"

let myName = "Gloria"                  "Gloria"
myName = "Matt"    Cannot assign to value: 'myName' is a 'let' constant   "Matt"
```

```
Playground execution failed: /var/folders/q3/9zwqk4ws2gvc6pl001j1c53w0000gn/T/./lldb/410/playground11.swift:12:8: error: cannot assign to
value: 'myName' is a 'let' constant
myName = "Matt"
~~~~~~ ^
/var/folders/q3/9zwqk4ws2gvc6pl001j1c53w0000gn/T/./lldb/410/playground11.swift:11:1: note: change 'let' to 'var' to make it mutable
let myName = "Gloria"
^~~
var
```

Figure 2-5: Trying to change the value of a constant won't work.

NOTE *In the playground, an error will appear as a red circle with a tiny white circle inside it. Clicking the error mark will show the error message and tell you what's wrong. If you have your debug area showing, you should also see information describing what happened and sometimes even how to fix it.*

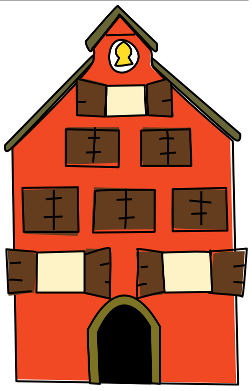
When to Use Constants vs. Variables

Now you've successfully created a variable and a constant—good job! But when should you use one over the other? In Swift, it is best practice to use constants instead of variables unless you expect that the value will change. Constants help make code "safer." If you know the value of something is never going to change, why not etch it into stone and avoid any possible confusion later?

For example, say you want to keep track of the total number of windows in your classroom and the number of windows that are open today. The number of windows in your classroom isn't going to change, so you should use a constant to store this value. The number of windows that are open in your classroom will change depending on the weather and time of day, however, so you should use a variable to store this value.

```
let numberOfWindows = 8                8
var numberOfWindowsOpen = 3            3
```

Here we make `numberOfWindows` a constant and set it to `8` because the total number of windows will always be 8. We make `numberOfWindowsOpen` a variable and set it to `3` because we'll want to change that value when we open or close any windows.

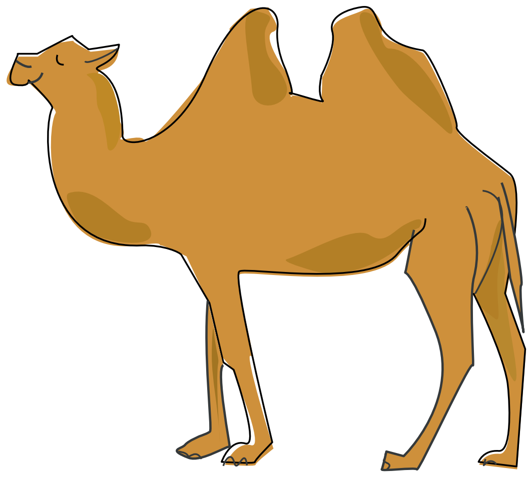


Remember: use `var` for variables and `let` for constants!

Naming Constants and Variables

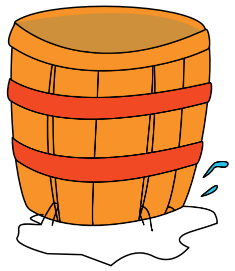
You can name a variable or constant almost anything you want, with a few exceptions. You can't name them something that is already a word in Swift. For example, you can't name a variable `var`. Writing `var var` would just be confusing, to you and the computer. You will get an error if you try to name a variable or constant using one of Swift's reserved words. You also can't have two variables or constants with the same name in the same block of code.

In addition to these rules, there are some other good programming guidelines to follow when naming things in Swift. Your names should always start with a lowercase letter. It's also a good idea to have *very* descriptive variable and constant names (they can be as long as you want). When you use a descriptive name, it's a lot easier to figure out what that variable or constant is supposed to be. If you were looking at someone else's code, which variable name would you find easier to understand: `numKids` or `numberOfKidsInMyClass`? The first one is vague, but the second one is descriptive. It is quite common to see variables and constants that are a bunch of words strung together, like `numberOfKidsInMyClass`. This capitalization style, where the first letter of each word is capitalized when multiple words are joined together to make a variable name, is called *camel case*. That's because the pattern of lowercase and uppercase letters looks like the humps on a camel's back.



DATA TYPES

In Swift, you can choose what kind of data—the *data type*—you want a variable or constant to hold. Remember how we said you can think of a variable as a container that holds something? Well, the data type is like the container type. The computer needs to know what kind of things we will be putting in each container. In Swift programming, once you tell the computer you want a variable or constant to hold a certain data type, it won't let you put anything but that data type in that variable or constant. If you have a basket designed to hold potatoes, it'd be a bad idea to fill that basket with water—unless you like water leaking all over your shoes!



Declaring Data Types

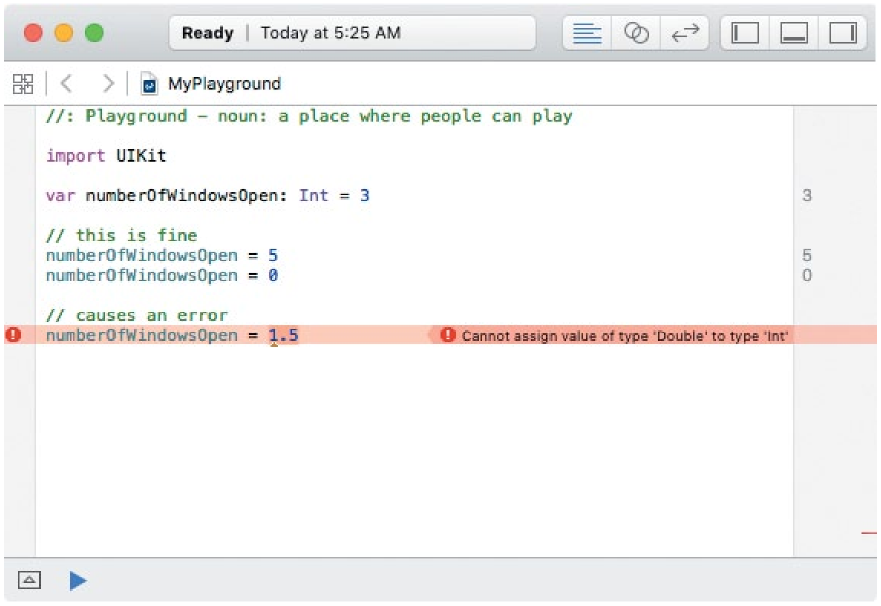
When you create a variable or a constant, you can tell the computer what type of data it will hold. In our example about classroom windows, we know this variable will always be a whole number (you can't really have half a window), so we could specify an *integer* data type, like this:

```
var numberOfWindowsOpen: Int = 3    3
```

The colon means "is of type." In plain English, this line of code says, "the variable `numberOfWindowsOpen`, which is an integer, is equal to 3." So this line of code creates a variable, gives it a name, tells the computer what sort of data it will hold, and then assigns it a value. Phew! One line of code did all that? Did we mention that Swift is a very *concise* language? Some languages might require several lines of code to do this same thing. Swift is designed so that you can do a bunch of things with just one line of code!

You only have to declare the data type once. When we tell the computer that a variable will hold integers, we don't have to tell it again. In fact, if we try to do that, Xcode will give us an error. Once the data type is declared, a variable or constant will hold that same type of data forever. Once an integer, always an integer!

There's one more thing you need to know about data types: a variable or constant cannot hold something that is not its data type. For example, if you try to put a decimal number into `numberOfWindowsOpen`, you'll get an error, as shown in Figure 2-6.



```
//: Playground - noun: a place where people can play

import UIKit

var numberOfWindowsOpen: Int = 3                                   3

// this is fine
numberOfWindowsOpen = 5                                            5
numberOfWindowsOpen = 0                                            0

// causes an error
numberOfWindowsOpen = 1.5        Cannot assign value of type 'Double' to type 'Int'
```

Figure 2-6: You can't put a decimal number into a variable that is supposed to hold an integer.

Setting `numberOfWindowsOpen = 5` and `numberOfWindowsOpen = 0` is valid and works. But you can't set `numberOfWindowsOpen = 1.5`.

Common Data Types

As you just learned, a data type lets the computer know what *kind* of data it is working with and how to store it in its memory. But what are the data types? Here are some common ones that you'll be working with:

- `Int`
- `Double`
- `Float`

- `Bool`
- `String`

Let's dig in and see what each one of these actually is!

Int (Integers)

We already talked a little bit about integers, but let's go over them in more detail. An *integer*, called an `Int` in Swift, is a whole number that has no decimal or fractional part. You can think of them as counting numbers. Integers are *signed*, meaning that they can be negative or positive (or zero).

Double and Float (Decimal Numbers)

Decimal numbers are numbers that have digits after the decimal point, like 3.14. (An integer like 3 would be written as 3.0 if you wanted it to be a decimal number.) There are two data types that can store decimal numbers: a `Double` and a `Float`. `Double`s are more common in Swift because they can hold bigger numbers, so we'll focus on those.

When you assign a `Double`, you must always have a digit to the left of the decimal place or you will get an error. For example, suppose bananas cost 19 cents each:

```
❶ var bananaPrice: Double = .19 // ERROR
❷ var bananaPrice: Double = 0.19 // CORRECT
```

```
0.19
```

The code at ❶ will result in an error because it doesn't have a digit to the left of the decimal point. The code at ❷ works fine because it has a leading zero.

Bool (Booleans, or True/False)

A *Boolean value* can only be one of two things: true or false. In Swift, the Boolean data type is called a `Bool`.

```
let swiftIsFun = true
var iAmSleeping = false
```

```
true
false
```

`Bools` are often used in `if-else` statements to tell the computer which path a program should take. (We'll cover `Bools` and `if-else` statements in more detail in Chapter 3.)

String

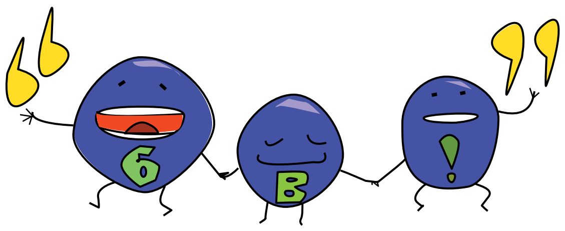
The `String` data type is used to store words and phrases. A *string* is a collection of characters enclosed in quotation marks. For example, `"Hello, playground"` is a string. Strings can be made up of all sorts of characters: letters, numbers, symbols, and more. The quotation marks are important because they tell the computer that everything in between the quotes is part of a string that you're creating.

You can use strings to build sentences by adding strings together in a process called string *concatenation*. Let's see how it works! Try this in your playground:

```
let morningGreeting = "Good Morning"
let friend = "Jude"
let specialGreeting = morningGreeting + " " + friend
```

```
"Good Morning"
"Jude"
"Good Morning Jude"
```

By adding strings together with the plus sign (+), this code creates a variable called `specialGreeting` with the string `"Good Morning Jude"` as its value.



Type Inference

You may have noticed that sometimes when we declare a variable, we include the data type:

```
var numberOfWindowsOpen: Int = 3
```

```
3
```

And sometimes we do not include the data type:

```
var numberOfWindowsOpen = 3                3
```

What gives? The computer is actually smart enough to figure out the data type, most of the time. This is called *type inference*—because the computer will *infer*, or guess, the type of data we are using based on clues that we give it. When you create a variable and give it an initial value, that value is a big clue for the computer. Here are some examples:

- If you assign a number with no decimal value (like 3), the computer will assume it's an `Int`.
- If you assign a number with a decimal value (like 3.14), the computer will assume it's a `Double`.
- If you assign the word *true* or *false* (with no quotes around it), the computer will assume it's a `Bool`.
- If you assign something with quotes around it, the computer will assume it's a `String`.

When the type is inferred, the variable or constant is set to that data type just as if you had declared the data type yourself. This is done purely for convenience. You can include the data type every time you declare a new constant or variable, and that's perfectly fine. But why not let the computer figure it out and save yourself the time and extra typing?

Casting

Casting is a way to temporarily transform the data type of a variable or constant. You can think of this as casting a spell on a variable—you make its value behave like a different data type, but just for a short while. To do this, you write a new data type followed by parentheses that hold the variable you are casting. Note that this *doesn't actually change the data type*, it just gives you a temporary value for that one line of code. Here are a few examples of casting between `Int` and `Double`. Take a look at the results of your code in the results sidebar.

```
  let months = 12
  print(months)
❶ let doubleMonths = Double(months)
  print(doubleMonths)
```

```
12
"12\n"
12
"12.0\n"
```

At ❶, we cast our `Int` variable `months` to a `Double` and store it in a new variable called `doubleMonths`. This adds a decimal place, and the result of this casting is `12.0`.

You can also cast a `Double` to an `Int`:

```
  let days = 365.25
❶ Int(days)
```

```
365.25
365
```

At ❶, we assign we cast our `Double`, `days`, to an `Int`. You can see that the decimal place and all the digits following it were removed: our number became `365` when cast to an `Int`. This is because an `Int` is not capable of holding a decimal number—it can contain only whole numbers, as we learned earlier. So anything after the decimal point is chopped off.

Again, casting does not actually change a data type. In our example, even after casting, `days` is *still* a `Double`. We can verify this by printing `days`:

```
print(days)
```

```
"365.25\n"
```

In the results sidebar, you'll see that `days` is still equal to `365.25`.

In the next section, we'll cover some examples of where and when you would use casting. So if it's not clear right now why you would cast a variable, just hold on a bit longer!

OPERATORS

There are a number of arithmetic operators in Swift that you can use to do math. You have already seen the basic assignment operator, `=`. You are probably also familiar with what these four operators do:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division

You can use these operators to perform math on `Int`s, `Float`s, and `Double`s. The numbers being operated on are called *operands*. Experiment with these mathematical operators in your playground by writing code like the following:

```
6.2 + 1.4                       7.6
3 * 5                           15
16 - 2                          14
9 / 3                           3
```

If you type this code in your playground, you will see the results of each mathematical expression in the results sidebar. As you can see, writing mathematical expressions in code is not that different from writing them normally. For example, 16 minus 2 is written as `16 - 2`.

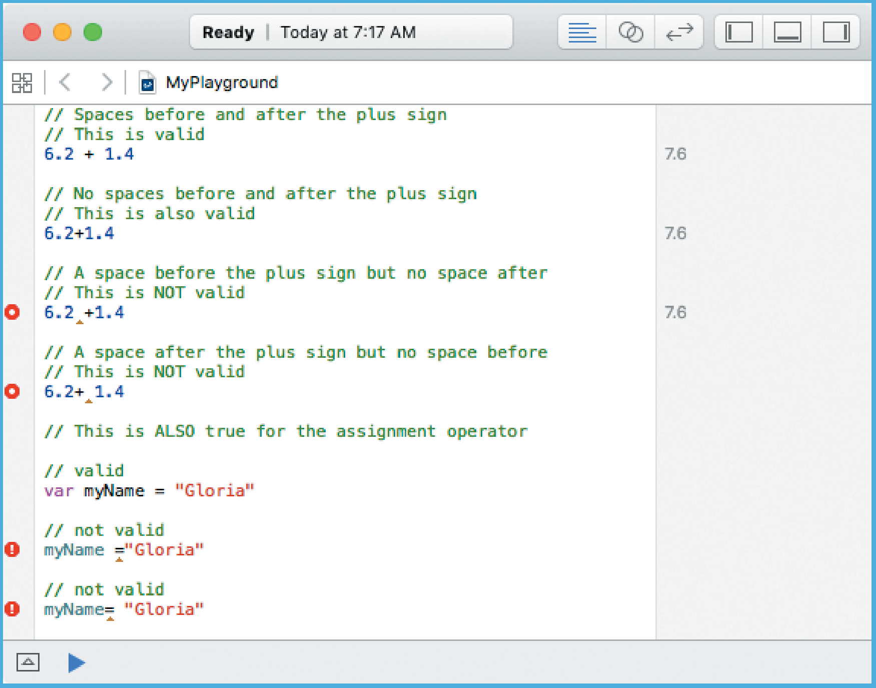
You can even save the result of a mathematical expression in a variable or constant so you can use it somewhere else in your code. To see how this works, type these lines in your playground:

```
  var sum = 6.2 + 1.4           7.6
❶ print(sum)                    "7.6\n"
  let threeTimesFive = 3 * 5    15
```

When you print `sum` ❶, you'll see the value `7.6` in the results sidebar.

SPACES MATTER

In Swift, the spaces around an operator are important. You can either write a blank space on both sides of the mathematical operator or leave out the spaces altogether. But you cannot just put a space on one side of the operator and not the other. That will cause an error, and it makes your code look messy. Take a look at Figure 2-7.



```
// Spaces before and after the plus sign
// This is valid
6.2 + 1.4                                          7.6

// No spaces before and after the plus sign
// This is also valid
6.2+1.4                                            7.6

// A space before the plus sign but no space after
// This is NOT valid
6.2 +1.4                                           7.6

// A space after the plus sign but no space before
// This is NOT valid
6.2+ 1.4

// This is ALSO true for the assignment operator

// valid
var myName = "Gloria"

// not valid
myName ="Gloria"

// not valid
myName= "Gloria"
```

Figure 2-7: Make sure that you have the same number of spaces on each side of your operators.

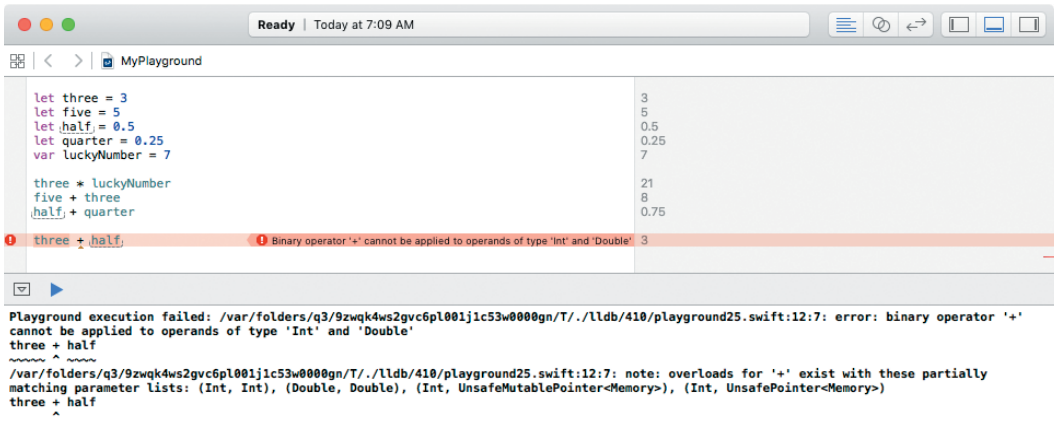
So far, we have used only numbers in our mathematical expressions, but mathematical operators will also work on variables and constants.

Add the following code to your playground:

```
let three = 3                      3
let five = 5                       5
let half = 0.5                     0.5
let quarter = 0.25                 0.25
var luckyNumber = 7                5

three * luckyNumber                21
five + three                       8
half + quarter                     0.75
```

As you can see, you can use the mathematical operators on variables and constants in the same way you did on numbers.

There is one important thing to note: you can only use a mathematical operator on two variables or constants that are the *same* data type. In the previous code, `three` and `five` are both `Int`s. The constants `half` and `quarter` are `Double`s because they are decimal numbers. If you try to add or multiply one of the `Int`s and one of the `Double`s, you'll get an error like the one in Figure 2-8.



```
let three = 3                      3
let five = 5                       5
let half = 0.5                     0.5
let quarter = 0.25                 0.25
var luckyNumber = 7                7

three * luckyNumber                21
five + three                       8
half + quarter                     0.75

three + half     Binary operator '+' cannot be applied to operands of type 'Int' and 'Double'   3
```

```
Playground execution failed: /var/folders/q3/9zwqk4ws2gvc6pl001j1c53w0000gn/T/./lldb/410/playground25.swift:12:7: error: binary operator '+'
cannot be applied to operands of type 'Int' and 'Double'
three + half
~~~~~ ^ ~~~~
/var/folders/q3/9zwqk4ws2gvc6pl001j1c53w0000gn/T/./lldb/410/playground25.swift:12:7: note: overloads for '+' exist with these partially
matching parameter lists: (Int, Int), (Double, Double), (Int, UnsafeMutablePointer<Memory>), (Int, UnsafePointer<Memory>)
three + half
      ^
```

Figure 2-8: In Swift, you cannot do math on mixed data types.

But what if you really want to do math on mixed data types? For example, let's say you want to calculate one-tenth of your age:

```
var myAge = 11 // This is an Int                    11
let multiplier = 0.1 // This is a Double            0.1
var oneTenthMyAge = myAge * multiplier
```

The last line will result in an error because we are attempting to multiply an `Int` by a `Double`. But don't worry! You have a couple of options to make sure your operands are the same data type.

One option is to declare `myAge` as a `Double`, like this:

```
var myAge = 11.0 // This is a Double                11.0
let multiplier = 0.1 // This is a Double            0.1
var oneTenthMyAge = myAge * multiplier              1.1
```

This code works because now we're multiplying two `Double`s.

The second option is to use casting (I told you we would come back to this!). Let's take a look at an example:

```
  var myAge = 11 // This is an Int                      11
  let multiplier = 0.1 // This is a Double              0.1
❶ var oneTenthMyAge = Double(myAge) * multiplier        1.1
❷ oneTenthMyAge = myAge * multiplier
```

At ❶, we cast `myAge` to a `Double` before multiplying it. This means we no longer have mixed types, so the code works. But at ❷ we will get an error. That's because `myAge` is still an `Int`. Casting it to a `Double` at ❶ did not permanently change it to a `Double`. Casting is a great solution in this case because we don't want to permanently change `myAge` to a `Double`, we just want to be able to perform math with it as if it were a `Double`.

Could we cast `multiplier` to an `Int`? You bet! Then we are doing math on two integers, which works fine. However, this results in a less precise calculation because we'll lose the decimal place. When you cast a variable from a `Double` to an `Int`, the computer simply removes any digits after the decimal to make it a whole number. In this case, your `multiplier` of `0.1` would cast to an `Int` of `0`. Let's cast some variables in the playground and see what we get:

```
❶ Double(myAge)          11
❷ Int(multiplier)        0
❸ Int(1.9)               1
```

At ❶, casting our `Int`, `myAge`, to a `Double` gives us `11.0`. So the value hasn't changed, but it now has a decimal place. At ❷, casting our `Double`, `multiplier`, to an `Int` gives us `0`. This value is quite different after casting, because we lost the decimal place: `0.1` became `0`. This could be a very bad thing in our code if we were not expecting it to happen. You must be careful when casting to make sure you aren't unexpectedly changing your values. At ❸, there's another example of casting a `Double` to an `Int`, and as you can see, `1.9` does not get rounded up to `2`. Its decimal value just gets removed and we are left with `1`.

There's another mathematical operator, the *modulo operator* (written as `%`), which might not be as familiar to you. The modulo operator (also called *modulus*) gives the remainder after division. For example, `7 % 2 = 1` because 7 divided by 2 has a remainder of 1. Try out the modulo operator with these examples in your playground:

```
  10 % 3                  1
  12 % 4                  0
  34 % 5                  4

  var evenNumber = 864    864
❶ evenNumber % 2          0

  var oddNumber = 571     571
❷ oddNumber % 2           1
```

As you can see, the modulo operator is useful for determining whether a number is even (`x % 2` equals 0) ❶ or odd (`x % 2` equals 1) ❷.

Order of Operations

So far we've only done one mathematical operation on each line of code, but it's common to do more than one operation on a single line. Let's look at an example.

You have three five-dollar bills and two one-dollar bills. How much money do you have? Let's do this calculation on one line of code:

```
var myMoney = 5 * 3 + 2                        17
```

This assigns a value of `17` to `myMoney`. The computer multiplies 5 times 3 and then adds 2. But how does the computer know to multiply first and *then* add 2? Does it just work from left to right? No! Take a look at this:

```
myMoney = 2 + 5 * 3                            17
```

We moved the numbers around and the result is still 17. If the computer just went from left to right, it would add 2 + 5 and get 7. Then it would multiply that result, 7, times 3, and get 21. Even though we changed the order of the numbers in our mathematical expression, the computer still does the multiplication first (which gives us 15) and then adds the 2 to get 17. *The computer will always do multiplication and division first, then addition and subtraction.* This is called the *order of operations*.

Parentheses

You don't have to rely on the computer to figure out which step to do first like we did in the money example. You, the programmer, have the power to decide! You can use parentheses to group operations together. When you put parentheses around something, you tell the computer to do that step first:

```
❶ myMoney = 2 + (5 * 3)                          17
❷ myMoney = (2 + 5) * 3                          21
```

At ❶, the parentheses tell the computer to multiply 5 times 3 first and then add 2. This will give you 17. At ❷ the parentheses tell the computer to add 2 plus 5 first and then multiply that by 3, which gives you 21.

You can make your code even more specific by using parentheses inside of other parentheses. The computer will evaluate the inside parentheses first, then the outside ones. Try this example:

```
myMoney = 1 + ((2 + 3) * 4)                21
```

First the computer adds 2 and 3 because that's between the inner set of parentheses. Then it multiplies the result by 4, since that's within the outer set of parentheses. It will add the 1 last because it's outside of both sets of parentheses. The final result is 21.

Unary Operators

So far, the arithmetic operators we've looked at require two numbers. But there are three operators that operate on a single number. These are called *unary* operators:

`-`	Negation
`++`	Increment
`--`	Decrement

The first unary operator we'll cover is *negation*. The minus sign (-) negates a value, and it works for both numbers and variables, like `-10` or `-y`.

```
  var myNumber = 534                              534
  var myNegativeNumber = -myNumber                -534
  var myOtherNumber = -245                        -245
❶ var myPositiveNumber = -myOtherNumber           245
```

As you can see at ❶, negation returns a positive value if you use it on a negative number.

The increment operator (`++`) *increments*, or increases, the value of the operand by 1. It is written as `++x`, which can be read as "add one to x." Similarly, the decrement operator (--) *decrements*, or decreases, the value by 1. It is written as --x, which can be read as "subtract one from x." You will see the ++ and -- operators a lot in loops, which you'll learn about in Chapter 4.

Compound Assignment Operators

The final category of operators that you'll use is the *compound assignment operators*. These are "shortcut" operators that combine a mathematical operator with the assignment operator (=). For example, this expression

```
a = a + b
```

becomes

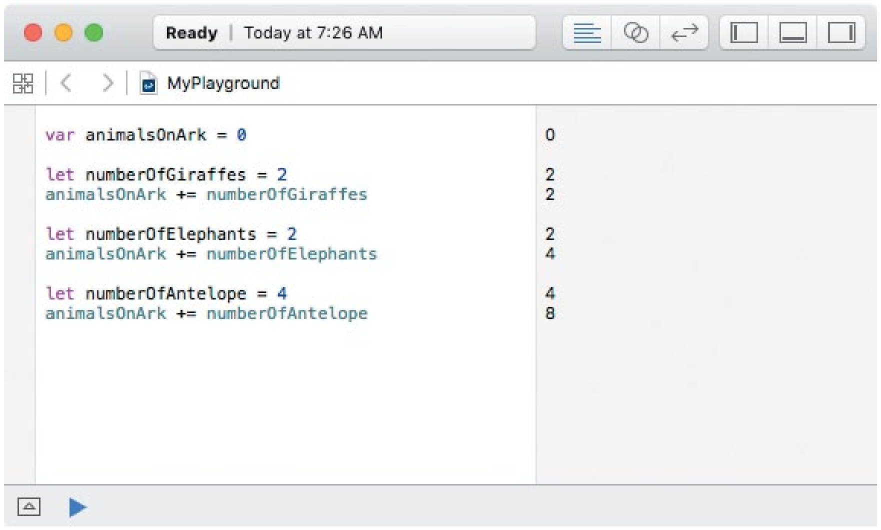
```
a += b
```

You can use these operators to update the value of a variable or constant by performing an operation on it. In plain English, an expression like a += b says "add b to a and store the new value in a." Table 2-1 shows mathematical expressions using compound assignment operators and the same expressions in their longer forms.

Table 2-1: Expressions Using Compound Assignment Operators vs. Expressions in Long Form

Short form	Long form
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`

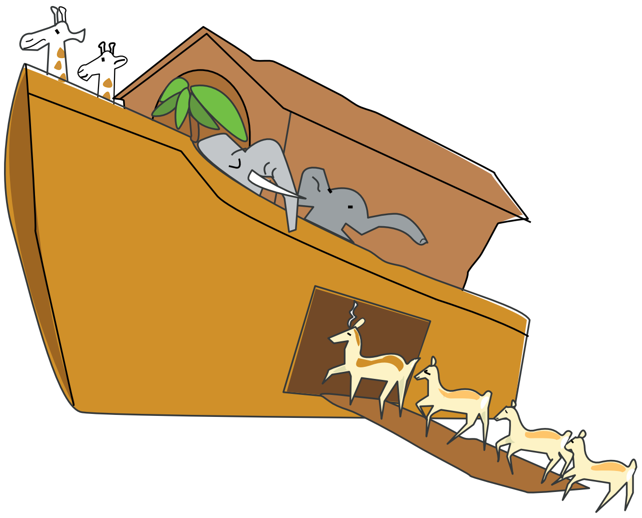
Let's watch these operators in action. Imagine that we're trying to write a program to calculate the number of animals on an ark. First we create a variable called animalsOnArk and set it to 0 because there aren't animals on the ark yet. As the different types of animals board the ark, we want to increase `animalsOnArk` to count all of the animals. If two giraffes board the ark, then we need to add 2 to `animalsOnArk`. If two elephants board the ark, then we need to add 2 again. If four antelopes board the ark, then we need to increase `animalsOnArk` by 4. You can see this code in Figure 2-9.



```
var animalsOnArk = 0                              0

let numberOfGiraffes = 2                          2
animalsOnArk += numberOfGiraffes                  2

let numberOfElephants = 2                         2
animalsOnArk += numberOfElephants                 4

let numberOfAntelope = 4                          4
animalsOnArk += numberOfAntelope                  8
```

Figure 2-9: Using a compound assignment operator to tally `animalsOnArk`

After the giraffes, elephants, and antelopes board the ark, the final value for `animalsOnArk` is `8`. What a zoo!



A FEW QUICK COMMENTS ABOUT COMMENTS

Most programming languages come with a way to write *comments* directly inline with the code. Comments are notes added to the code that are ignored by the computer and are there to help the humans reading the code understand what's going on. Although a program will run completely fine without any comments, it's a good idea to include comments for sections of code that might be unclear or confusing. Even if you're not going to show your program to anybody else, your comments will help you remember what you were doing or thinking when you wrote that code. It's not uncommon to come back to a piece of code you wrote months or years ago and have no idea what you were thinking at the time.

There are two ways to add comments to code in Swift. The first way is to put two forward slashes (//) in front of the text you want to add. These comments can be placed on their own line, like this:

```
 // My favorite things
```

Or they can be placed on the same line as a line of code—as long as the comment comes *after* the code:

```
var myFavoriteAnimal = "Horse" // does not have to be a pet
```

The second way of adding comments is used for long comments, or *multiline* comments, where the start and end of the comment is marked by /* and */.

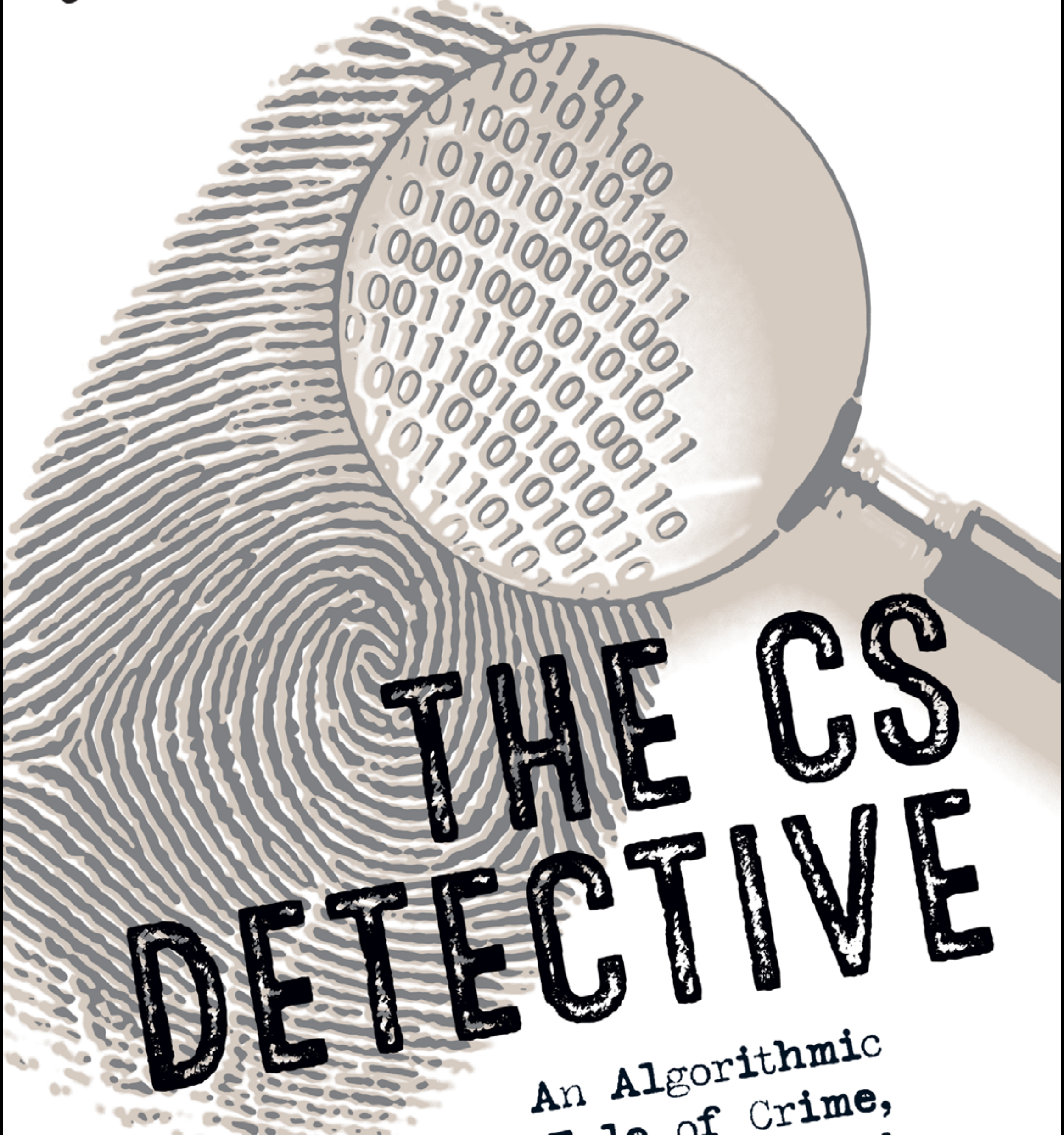
```
/*
  This block of code will add up the animals
  that walk onto an ark.
*/
{
  var animalsOnArk = 0
  let numberOfGiraffes = 2
  animalsOnArk += numberOfGiraffes
  --snip--
}
```

Multiline comments are also very useful when you are debugging your code. For example, if you don't want the computer to run some part of your code because you're trying to find a bug, but you also don't want to delete all of your hard work, you can use multiline comments to *comment out* large sections of code temporarily. When you format a chunk of code as a comment, the computer will ignore that code just like it ignores any other comment.

WHAT YOU LEARNED

In this chapter, you learned how to write code in a Swift playground, which lets you see results right away. You learned how to create variables and constants. You also learned the basic data types and operators that you'll be using when writing your own computer programs.

jeremy kubica



THE CS DETECTIVE

An Algorithmic Tale of Crime, Conspiracy, and Computation

— 1 — Search Problems

The door opened without a knock—only the hinge's creak announced the visitor. Frank started for his crossbow, but pulled up short. If the Vinettees were coming for him, they would have knocked—with an axe. Whoever was coming through the door must want to talk. Frank reached for his mug instead and downed the remainder of his now-cold coffee.

"Captain Donovan," he said as the man entered. "What brings you to this fine neighborhood? I thought you didn't venture below Fifteenth Street anymore."

"It's been a while," the captain said simply. "How've you been, Frank?"

"Spectacular," Frank answered dryly, eyeing the captain as he walked a slow circuit around the room.

Donovan scanned Frank's shabby office. His red officer's cloak swished gently behind him. "How's the private eye game?"

"It pays the bills," Frank lied.

The captain nodded. He paused for a moment, then moved to the bookshelf and browsed the contents.

"So is this a social visit then?" Frank said. "Should I be asking after Marlene and the kids?"

"They're quite well," replied Donovan without turning around. "Marlene's turtle-grooming business is doing well these days. Bill joined the force last year. And Veronica is an accountant, just about the last thing we would have—"

"I wasn't actually asking," Frank interrupted.

The captain shrugged. He pulled a book from the shelf and leafed through the pages. Frank craned his neck to see the cover—*Police Academy Yearbook: Class XXI*.

"What do you want, Captain?" Frank demanded.

The captain met Frank's stare at last. "I need your help, Frank," he said.

Frank straightened. In the five years since Frank had left the force, the captain had paid him exactly two visits, and both had been to warn him to stay away from active cases. Threats were all Frank had come to expect, but now it seemed the captain had a special kind of problem—perhaps the kind that would mean an end to Frank's delinquent rent.

"I'm not on the force anymore," said Frank airily. "Why don't you get one of your trusted detectives to do it?"

"I need someone outside of the force," said the captain. "Drop the act, Frank. If you don't know what it means for me to be here, you're not the person I need."

Frank chuckled. “A leak? On *your* force?”

“Worse. Last night someone broke into the station’s record room and stole over 500 scrolls.”

“What were they after?” asked Frank. Without thinking, he leaned forward in his chair and reached for a fresh scroll and a quill. The movement came automatically to him, like drinking coffee or avoiding stairs.

“I don’t know,” said Donovan. “There was no pattern. They stole whole shelves of documents, everything from property disputes to expense reports. They took all the ledgers we keep on assassins, celebrities, private investigators, notaries . . . They even took both boxes of Farmer Swinson’s noise complaints. But other shelves were completely untouched. We counted at least 512 missing documents.”

“Maybe it was one of Farmer Swinson’s neighbors,” joked Frank. “They must’ve heard that after a mere hundred complaints, an intern will come to your house and give you a stern lecture.”

Captain Donovan didn’t bother to reply. He just stared pityingly until Frank cleared his throat and broke the silence. “So you want me to find these documents?”

The captain shook his head. “I want you to find the thieves. We have backups of the documents. I want to know what information they needed and what they plan to do with it.”

“A search problem,” Frank mused. During his time on the force, his two specialties had been search problems and annoying the captain.

“Does the king know?” Frank asked.

“I briefed him yesterday,” said the captain, a hint of annoyance in his voice. “Ever since the trouble with that crackpot wizard, the king insists on daily briefings on everything.” Two years ago, a megalomaniac wizard named Exponentious had tried to destroy the entire kingdom. Since then King Fredrick had personally instituted sweeping upgrades to the kingdom's security, with over 300 new security regulations, at least 5 of which dealt with the storage of official documents in government buildings under 10 stories tall.

"I can't blame him though," Donovan grumbled. "It was a close call. If it hadn't been for Princess Ann, who knows where the kingdom would be now."

Frank nodded silently. Exponentious had attacked the algorithmic foundations of the kingdom by cursing the scholars who studied those algorithms. Within months he had rendered even simple operations inefficient, and the kingdom had started to grind to a halt. Evidence of the damage had been everywhere; even in his local bakery, Frank had himself witnessed panic break out as customers discovered they couldn't remember how to arrange themselves into a line.

"The king has, of course, taken a personal interest in the matter," the captain continued irritably. "He wants all the details: Who's assigned to the case? Which search algorithms are we using? Have we scoured all of the neighboring buildings?"

Frank stifled a chuckle and mulled over the proposition. A consulting gig for the capital's police force would be good money. He glanced down at his feet, where the tip of a toe peeked through a hole in his shoe. "If I'm going to consult," he said, "I'm going to do things my way."

This was the moment of truth. Five years ago he'd been kicked off the force for *doing things his way*. The captain was a man of rules and order. Frank's last use of heuristics had been the final straw—Captain Donovan had claimed his badge that very afternoon. But, then again, doing things his own way had always gotten Frank results.

"I figured as much," the captain responded at last. He pulled a thin folder from under his trench cloak and dropped it on Frank's desk.

"I'll be in touch," Donovan said. Then, without ceremony, he turned and left the office.

Three hours and twelve mugs of coffee later, Frank sat hunched over his desk and thumbed through the thin folder of information for the seventh time. The words jumped and swayed in the flickering candlelight, but didn't provide any new insights.

There wasn't a lot to go on. The captain had given him a list of missing documents and the duty roster for the night in question, but nothing more.

Finally, with an exaggerated sigh, Frank grabbed a piece of parchment and started making notes.

The first step in any search problem is determining what it is you hope to find—the *target*, as his old instructor in Police Algorithms 101 called it. Frank had learned that lesson early; he'd been tasked in his first week as an officer with finding the duke's prize stallion, and he'd proudly returned to the station that same afternoon with a 42-pound horned turtle. Apparently, the impressive reptile wasn't good enough. A good search algorithm means nothing if you're looking for the wrong thing.

In this case it wasn't a *what*, but rather a *who*. The captain had been right about that point. Once the thieves had the documents, it didn't matter if the police got them back. The thieves already had whatever information they needed.

So his target was simple: the person or persons who stole the documents.

The second step in any search problem is identifying the *search space*. What are you searching? During Frank's daily search for his keys, the search space was every flat surface in his office. And when

Frank wanted to find a criminal, his search space was every person in the vicinity of the capital.

Frank sat back and rubbed his eyes. It was a big search problem, finding a specific criminal in a city of criminals. But he had seen worse.

Now that he had defined the problem, he could start on an algorithm. A linear search was out; he couldn't afford to question everyone in the city. He could also rule out many of the other, fancier algorithms he had studied in the academy. For a problem like this, he would have to go back to his toolkit of basic search algorithms—the private investigator's most trusted friends.

Frank made a note on the parchment. He had the target to find, he knew the search space, and he had his algorithm. It was time to get to work.

POLICE ALGORITHMS 101: SEARCH PROBLEMS

Excerpt from Professor Drecker's Lecture

In this class we'll discuss several different algorithms (and related data structures) for solving search problems. A *search problem* is defined as any problem that requires us to find a specific value (or target) within a space of possible values (a search space).

Those of you who graduate and go on to become police officers will find yourselves facing problems that fall into this category every single day. This broad definition of a search problem encompasses a lot of different computational problems, from searching the police log for a specific entry to finding rooms within a hideout to finding all arrest records that match some criteria. This class won't be exhaustive—that would take years—but I'll give you some simple examples of basic and important algorithms as we go.

The algorithms described in this class will have three common components:

Target The piece of data you're searching for. The target can be either a specific value or a criterion that signifies the successful completion of a search.

Search space The set of all possibilities to test for the target. For example, the search space could be a list of values or all the nodes in a graph. A single possibility within the search space is called a *state*.

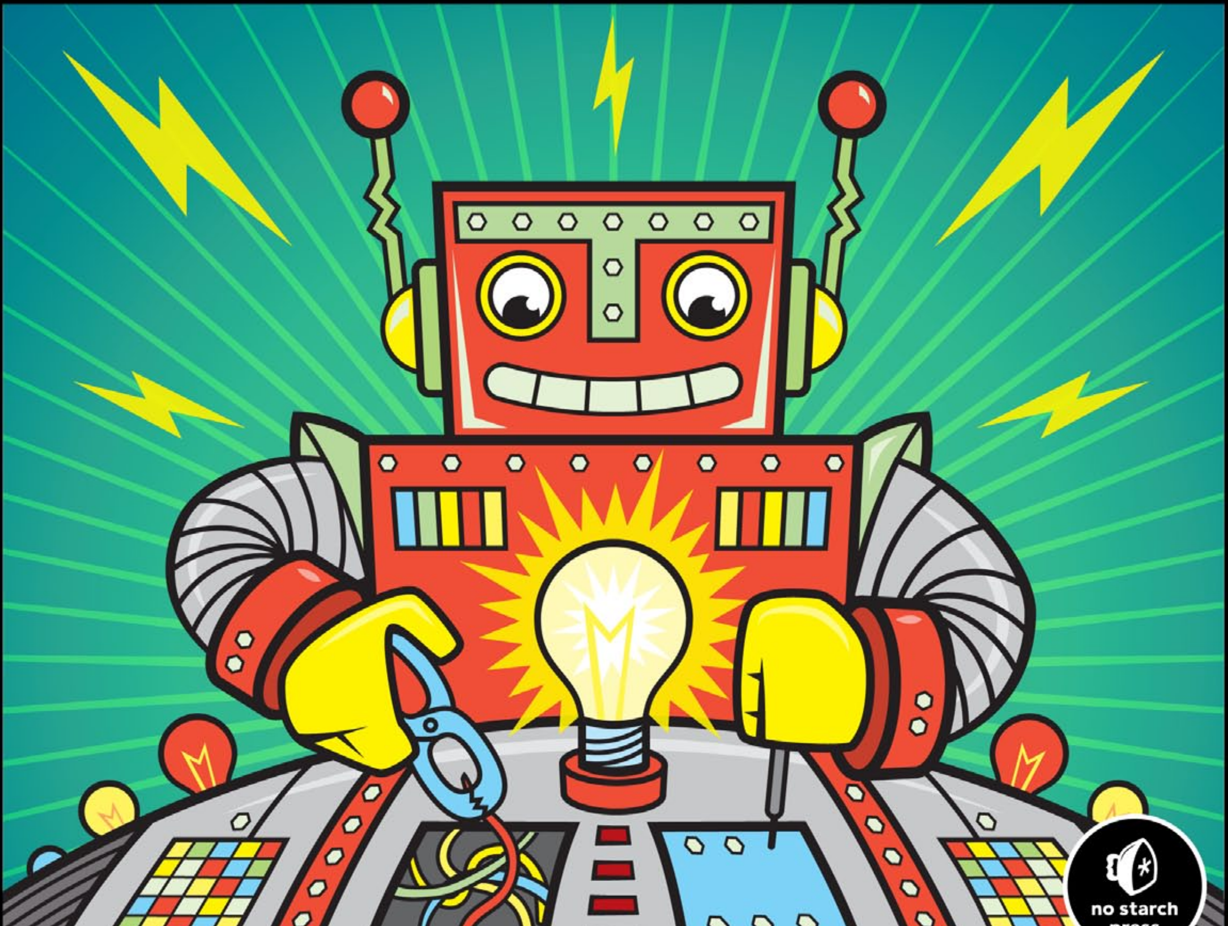
Search algorithm The set of specific steps or instructions for conducting the search.

Some search problems will have additional requirements or complexities, which we'll touch upon as we go over different algorithms.

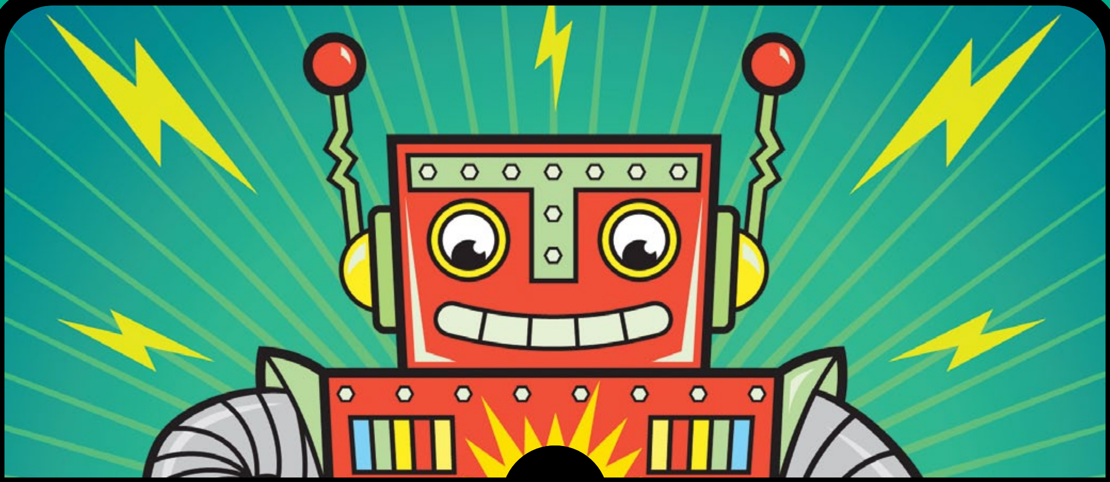
ELECTRONICS FOR KIDS

PLAY WITH SIMPLE CIRCUITS AND EXPERIMENT WITH ELECTRICITY!

ØYVIND NYDAL DAHL



no starch press



3 HOW TO GENERATE ELECTRICITY

Chapter 1 described why you need a closed loop to get current flowing through a circuit, and Chapter 2 showed you how to build your own electromagnet and motor. The projects in those chapters used electricity from a battery, but in this chapter, you'll make your own electricity sources!

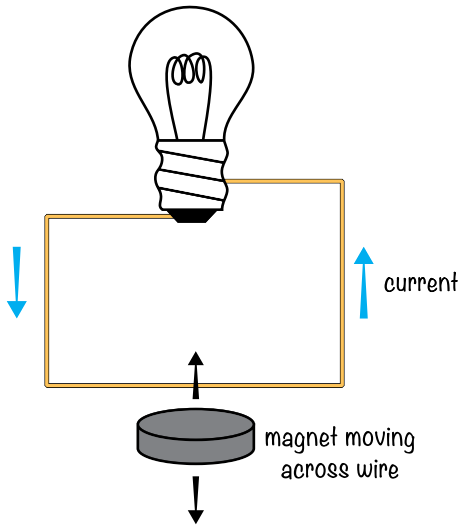
Specifically, you'll learn how to build your own generator, which creates electricity from movement, and your own battery, which creates electricity through chemical reactions. These are two of the most common ways to obtain electricity.

GENERATING ELECTRICITY WITH MAGNETS

When you run current through a wire, it creates a magnetic field around the wire, but there's another connection between electricity and magnetism. You can also create electricity using a wire and a magnet!

A Changing Magnetic Field Creates Electricity

If you move a magnet back and forth over a wire connected in a closed loop, you'll create a current in the wire. Moving the magnet changes the magnetic field around the wire, and the changing magnetic field pushes the electrons through the wire.

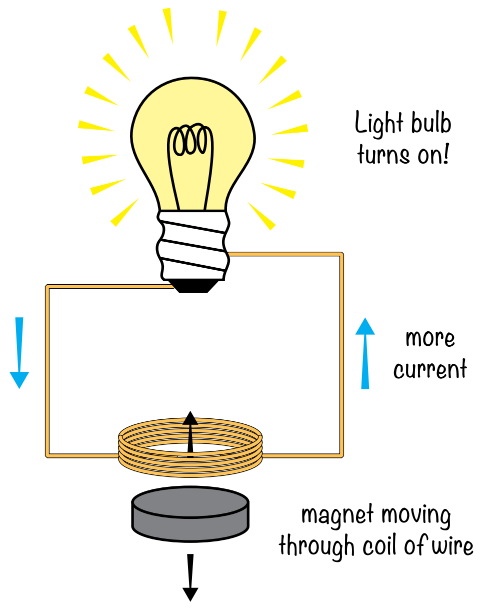


If you stop moving the magnet, the current also stops—even if the wire is still within the magnetic field—because the magnetic field is no longer changing.

If you connect the two ends of the wire to a light bulb and create a closed loop, then the current can flow. Unfortunately, however, the current created by moving a magnet over a single wire doesn't provide enough energy quickly enough to actually light the bulb. To light a bulb, or to power anything else, you need to find a way to generate more *power*, which is the amount of energy produced in a certain time.

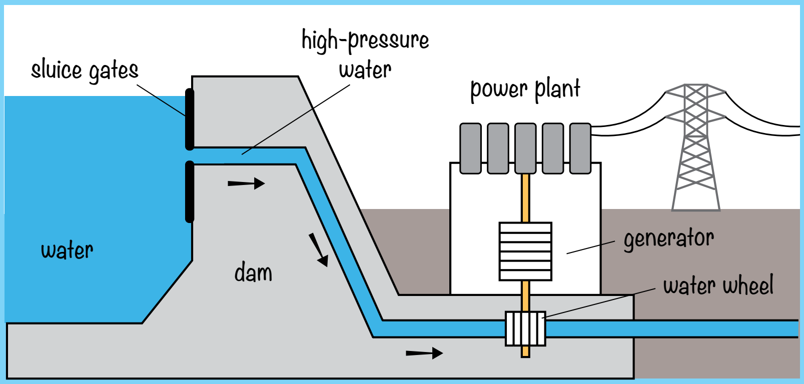
How Does a Generator Work?

A *generator* is a device that turns movement—such as the movement of a magnet over a wire—into electricity. To create more power with a wire and a magnet, you can wind that wire into a coil. The coiled wire acts like a group of wires, and when the magnetic field passes through it, a current flows through each coil, creating more power than you could with a straight wire.

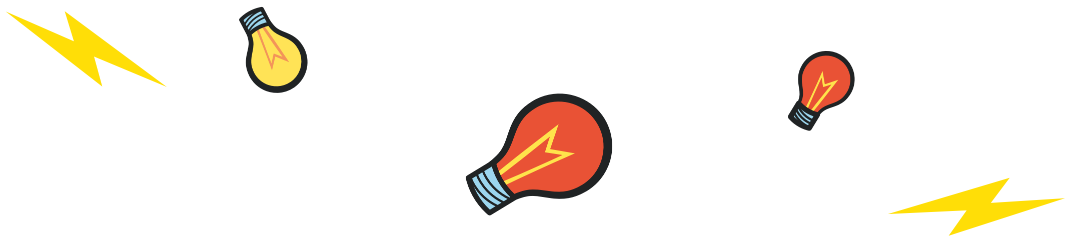


CREATING ELECTRICITY FROM WATER OR WIND

If you place a coil in a magnetic field and rotate the coil with a handle, you're converting your own movement into electricity. If you replaced the handle with a water wheel and placed it into a stream of water, the water would push the wheel so that the coil would rotate in the magnetic field and create a current. This is how some power plants generate electricity! The power plant just lets water run through a wheel that's connected to a generator. Then this electricity is transferred, through power lines, to the power outlets in people's houses.

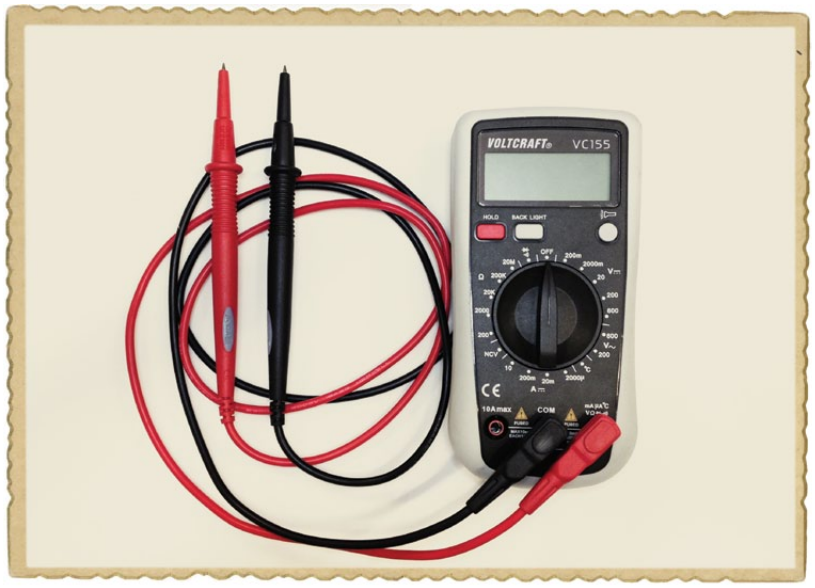


You can make electricity out of other natural forces in the same way. For example, to create electricity out of wind, you can connect the coil to a windmill so that when the wind blows, it rotates the coil.



MEET THE MULTIMETER

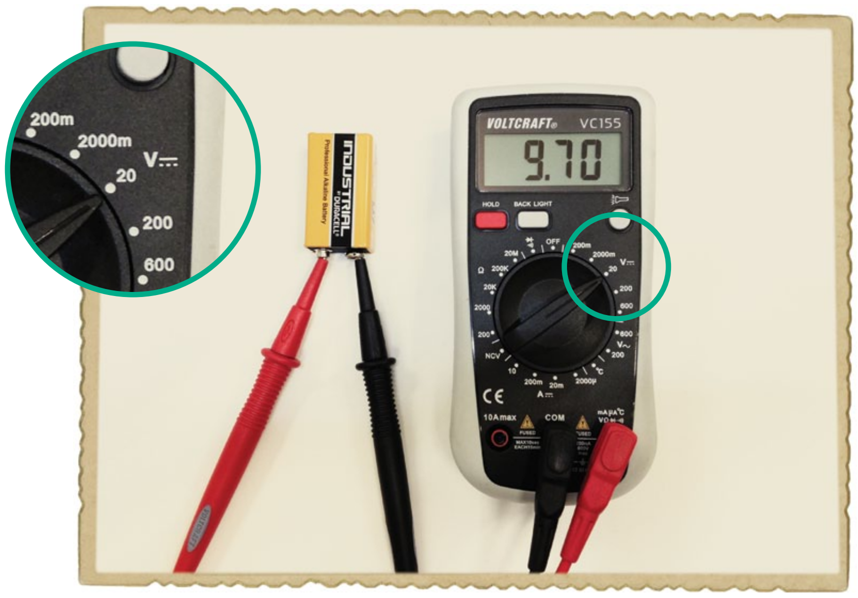
You can measure exactly how much energy a simple generator creates with a basic *multimeter*. Multimeters are handy when building any circuit because they can measure a lot of different values, including resistance, current, and voltage.



The red lead is the positive lead, the black lead is the negative lead, and the big dial in the middle lets you tell the multimeter what to measure. If you're having problems with a circuit, measuring the voltage at key points in your circuit is one practical way to figure out what's wrong.

How to Measure Voltage

To measure voltage with a multimeter, first turn the dial to one of the V options. (In this book, I'll tell you which setting to choose, but in your own projects, pick one that has a number higher than the highest voltage you expect to see in your circuit.) Then, at the bottom of the multimeter, connect the black lead to the COM socket and the red lead to the V socket. Finally, place one lead on each side of the part you want to measure the voltage across.



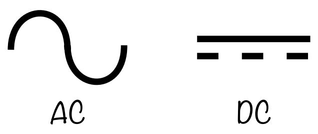
In this example, the meter is measuring the voltage between the positive and negative terminals of a 9 V battery. Notice that my dial is turned to 20 V, in the range showing a V with a straight-line symbol. But there's another V on the multimeter with a wavy line next to it. Let's look at what these symbols mean.

What Are AC and DC?

How you set your multimeter depends on whether you want to measure the voltage from a battery or a generator. A battery has a positive and a negative side, but a generator doesn't! A generator has two wires that alternate between being positive and negative. This is because when one side of the magnet moves past the coil, current in the coil flows in one direction, and when the other side of the magnet moves past the coil, current flows in the other direction.

When the current direction switches like that, we call it *alternating current (AC)*; when the direction of the current stays the same all the time, we call it *direct current (DC)*.

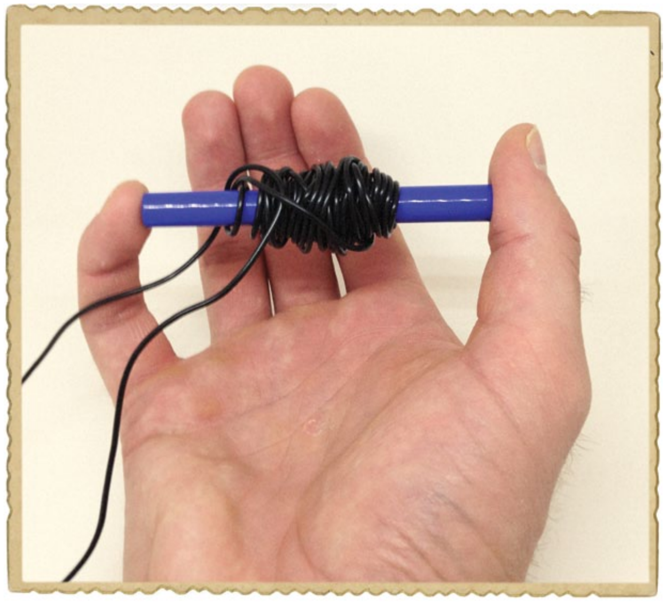
Usually, you'll find these symbols on your multimeter to indicate the AC and DC ranges of measurement:



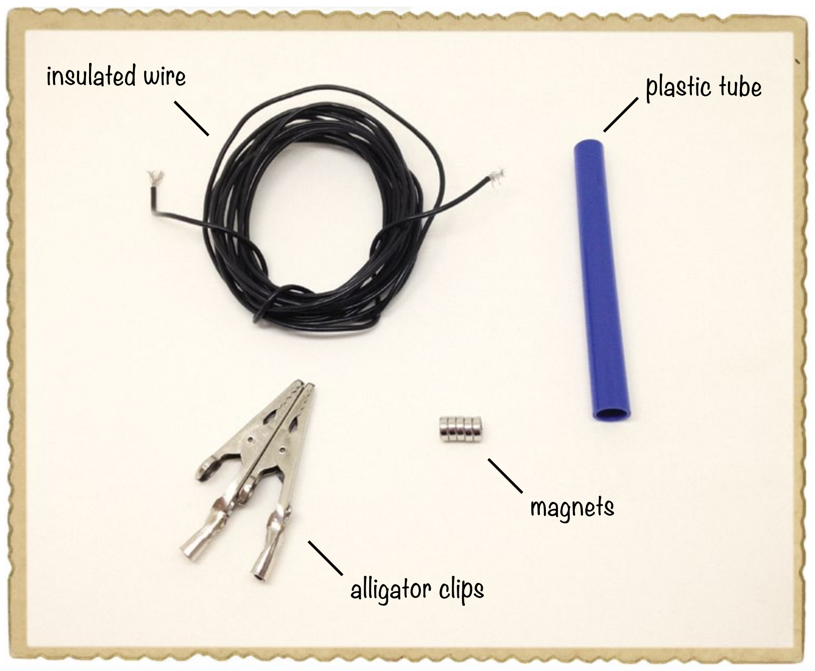
You need to set the multimeter to measure either AC or DC to get the correct reading. For example, batteries have a DC voltage.

PROJECT #5: MAKE A SHAKE GENERATOR

Grab your multimeter—this project will show you how to make a generator and measure its voltage. One quick way to create a simple generator is to manually move a magnet back and forth inside a coil. In this project, you'll put a magnet inside a tube and wind a coil around the tube. When you shake the tube, the magnet should move back and forth inside the coil and create a voltage.



Shopping List



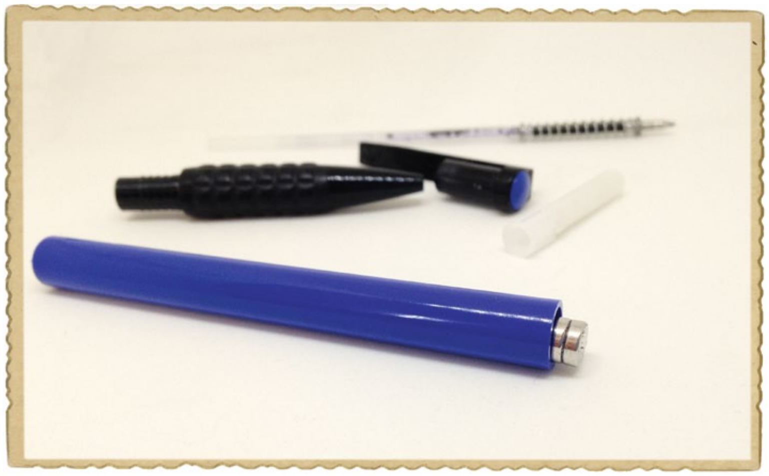
- **Insulated solid-core wire** (Jameco #36792, Bitsbox #W106BK), about 9 feet. Standard hookup wire works fine.
- **A small plastic tube**, such as an old pen.
- **Five disc magnets** (Jameco #2181319, Bitsbox #HW145) stacked to form a magnet rod.
- **Two alligator clips** (Jameco #256525, Bitsbox #CN262) to connect the multimeter to the coil.

Tools

- **A multimeter** to measure the voltage of your generator. The multimeter should be able to measure very low AC voltages, down to 0.01 V or less. Suitable multimeters are Jameco #2206061, Bitsbox #TL057, or Rapid Electronics #55-6662. These multimeters are a bit more expensive than the cheapest ones, but they will serve you for many years to come.

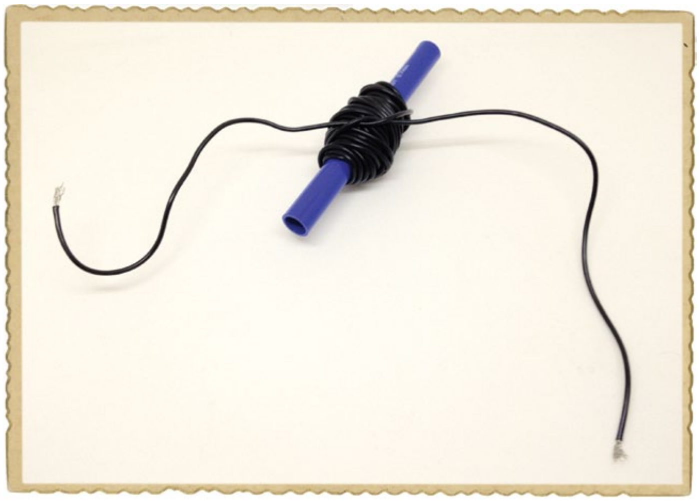
Step 1: Prepare Your Tube

Find a tube that's big enough to let the magnets slide easily back and forth. If you're using a pen, disassemble the pen and make sure your magnets fit inside the tube.



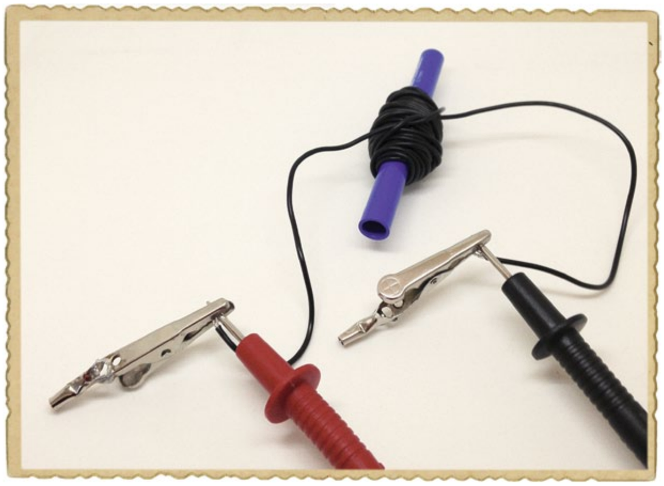
Step 2: Wind Your Coil

Wind about 50 turns of wire around the middle of your tube. After winding, make a simple knot with the two ends to keep your coil together. Then, strip the insulation from the two wire ends, as shown.



Step 3: Connect the Multimeter

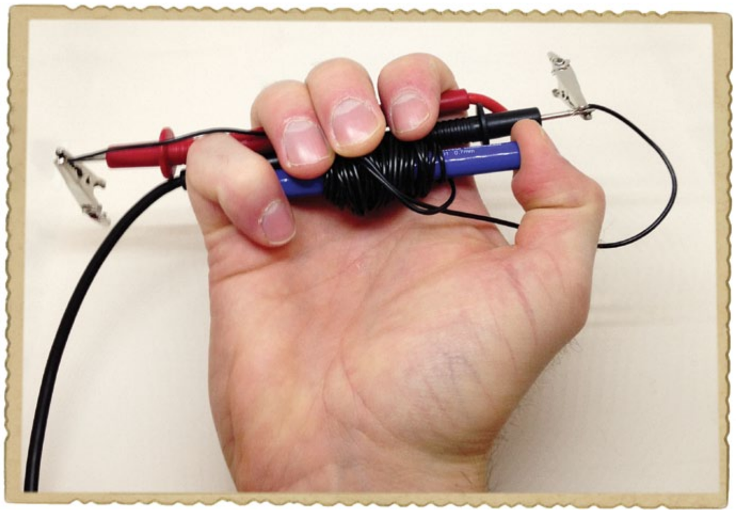
Connect the multimeter to both ends of the coil using alligator clips and set the multimeter to measure AC. Choose the lowest AC voltage setting available.



Step 4: Shake That Thing!

Next, put the magnets inside the tube. They should fit inside without coming apart.

Holding the tube and multimeter leads in your hand, place one finger on each side of the tube so that the magnets don't fall out. Then, shake it like you mean it!



Observe the voltage value on the multimeter. How much voltage do you get? I was able to get only 0.02 V from my generator, so it's not very powerful.

Step 5: What If There's No Voltage?

If you can't measure any voltage from your generator, first check that your multimeter leads are connected well to the exposed coil wires. If you still don't see a voltage higher than 0 V, make sure your multimeter is set to measure really low voltages; my dial was turned to 2 V AC. You won't get a high voltage from this simple generator, so if the multimeter isn't on the lowest setting possible, it will keep reading 0 V. Note that not all multimeters are able to measure such low voltages.

This generator isn't very powerful right now. How can you make it more powerful? Try to increase the voltage from the generator by shaking it faster, adding more loops of wire to the coil, or using a more powerful magnet.

NOTE *Standard hookup wire is a bit bulky; even 50 turns take up a lot of space! If you want to get a lot more turns, try using* magnet wire *instead. It's really thin wire with a thin layer of insulating coating.*

TRY IT OUT: USING A MOTOR AS A GENERATOR

A motor already has a magnet and a coil of wire that can rotate in the magnet's magnetic field. If you rotate the rotor with your hand, you can generate a voltage on the motor's wires.

You could create a generator by reversing the motor you built in Chapter 2, but the power you'd get from it would be too small to measure. Instead, try to find an old motor from a computer fan or a radio-controlled toy car that you don't want to play with anymore. Then, set your multimeter to a low-voltage DC range, such as 2 V DC. Attach the multimeter leads to the motor wires, just as you did with the shake generator, and turn the rotor with your fingers. Some motors have internal circuits that control the motor, and those circuits can prevent the electricity generated inside the motor from going out to the wires. But if you're lucky and find a motor that doesn't have such circuits, you should see a reading on the multimeter. Try a low-voltage AC range on your multimeter if you see nothing with DC.

HOW DO BATTERIES WORK?

I've shown you how to generate electricity manually, but that doesn't explain how you've powered circuits up to this point in the book. You've been using batteries, and in this section, we'll look at what lets those batteries create electricity.

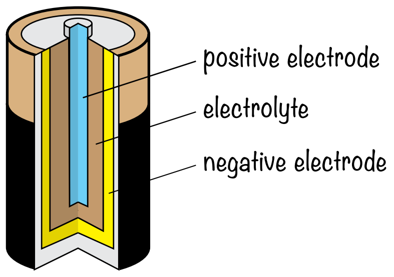
What's Inside a Battery?

To create a battery, you need three things:

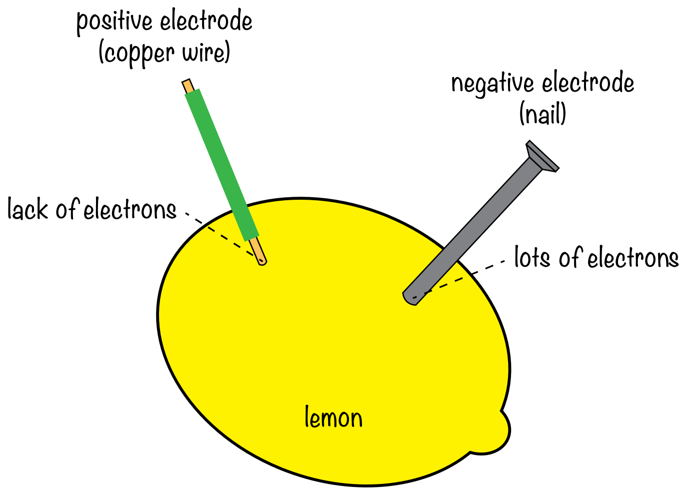
- A positive electrode
- A negative electrode
- An electrolyte

An *electrode* is a wire that is used to make contact with something nonmetallic, like the inside of a battery. An *electrolyte* is a substance that can release or gain electrons.

Here's how these three pieces fit inside a typical battery:



You can actually make your own battery by using a simple nail for one electrode and a copper wire for the other. Stick both into a lemon, and the lemon juice is your electrolyte.



The copper wire becomes the positive terminal of the battery, and the nail becomes the negative terminal.

The Chemistry Behind Batteries

When you combine the lemon, the copper wire, and the nail, two chemical reactions happen: one between the lemon juice and the nail, and another between the lemon juice and the copper wire. In the first reaction, electrons build up on the nail; in the second, electrons leave the copper wire. The nail gets too crowded with electrons, and the copper wire ends up with too few. Electrons don't like to be in crowded places, so the electrons on the nail want to go over to the copper wire to even things out. But the chemical reactions with the lemon juice are pushing the electrons the other way.

Now, what do you think will happen if you connect a light bulb between the nail and the copper wire? The electrons on the nail really want to get to the copper wire, so they'll take the easiest path they can find, and when you create this closed-loop circuit, they flow from the nail to the copper wire through the light bulb. Recall that current is just electrons flowing in a wire; if you have enough current flowing through the light bulb, it lights up!

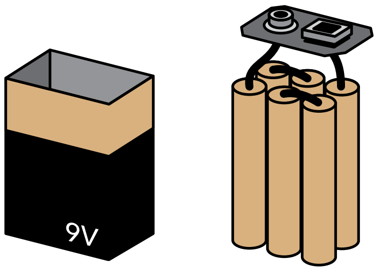
After a while, the chemical reactions in the battery stop. When this happens, the battery is dead. Some batteries can be recharged when they die, while others must be thrown away. The materials chosen for the electrodes and electrolyte determine whether the battery can be recharged or not.

The batteries you buy in the store are not made of lemons, of course! Modern batteries are made from different materials, and scientists are always looking for new ways to create batteries that have more energy, while being small and lightweight.

What Determines a Battery's Voltage?

The materials used for the electrodes and electrolyte determine the voltage you get from a battery, but the size of the electrodes and the amount of electrolyte don't matter when it comes to voltage.

To create higher battery voltages, several battery cells are connected in *series*. Connecting two battery cells in series means that you connect the positive side of one battery to the negative side of the other. The two unconnected terminals become the bigger battery's new positive and negative terminals, and the resulting voltage is the sum of the voltages from the two batteries. For example, in a standard 9 V battery, you have six 1.5 V battery cells, as shown. Notice that the connectors on the outside are attached to just two terminals.



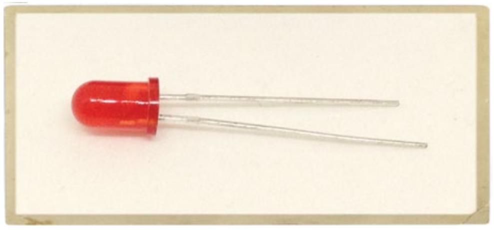
PROJECT #6: TURN ON A LIGHT WITH LEMON POWER

You can make a battery out of many different things; for example, in "What's Inside a Battery?" on page 89, I showed you how a lemon battery might work. In this project, you'll learn how to build a lemon battery of your own and power a light with it.

WARNING *When you're finished with this project, throw the lemons away. The chemical reactions that happen with the nail and copper wire will leave the lemons unsuitable for eating.*

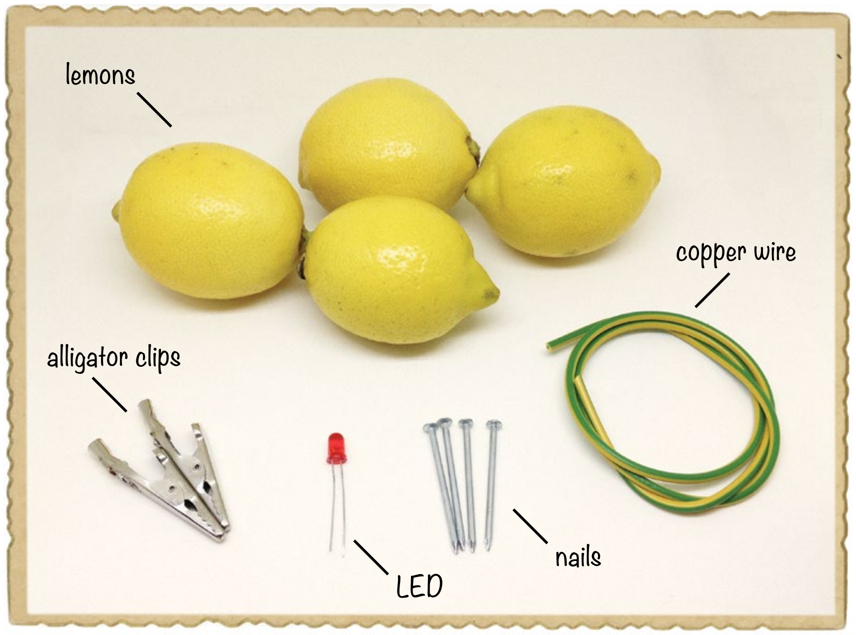
Meet the LED

A lemon battery can't create a lot of electricity, so you need to connect the battery to something that needs very little power to see the effect. Most light bulbs need more power than you'll generate in this project, so let me introduce a component called a *light-emitting diode*, or *LED*.



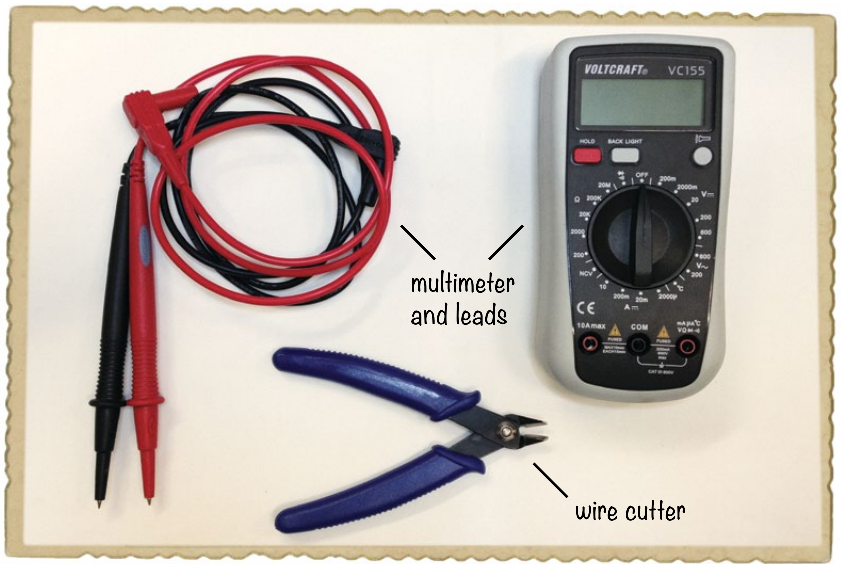
This little electronic component gives off, or *emits*, light when you apply a little bit of power to it. LEDs come in many colors: red, green, yellow, blue, and more. You'll learn more about this component in Chapter 4, and you'll use LEDs a lot in this book. For now, you're just going to use an LED to see the power generated by your lemon battery.

Shopping List



- **Four lemons** or one lemon cut into four pieces.
- **24 inches of copper wire** (any copper wire will do, but it's important that the wire be copper).
- **Four galvanized nails** (most common nails for outdoor projects are galvanized).
- **Two alligator clips** (Jameco #256525, Bitsbox #CN262) for connecting the LED.
- **A standard LED** (Jameco #333973, Bitsbox #OP002 for just this one, or Jameco #18041, Bitsbox #K033 for a variety pack). You'll need several LEDs for the projects in this book, so order at least 10 or a variety pack.

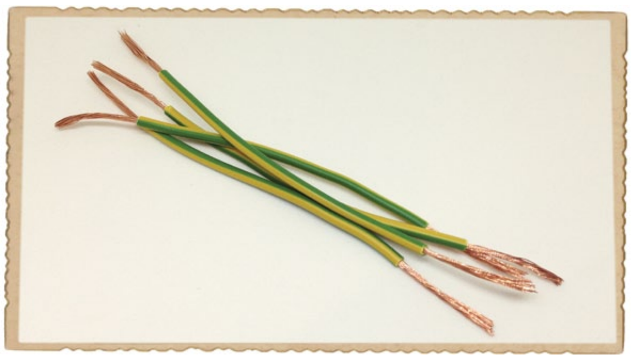
Tools



- **A wire cutter** (Jameco #35482, Bitsbox #TL008) to prepare the copper wire.
- **A multimeter** (Jameco #2206061, Bitsbox #TL057, Rapid Electronics #55-6662) to see whether your battery is working correctly.

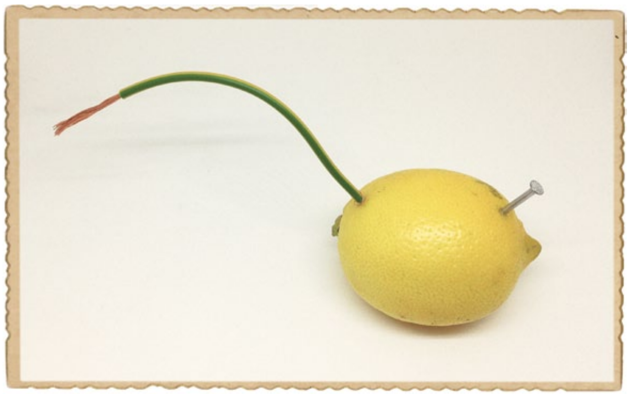
Step 1: Prepare Your Wires

First, cut your copper wire into four 6-inch lengths. Strip about 1 inch of insulation from both ends of each wire. These will become the electrodes.

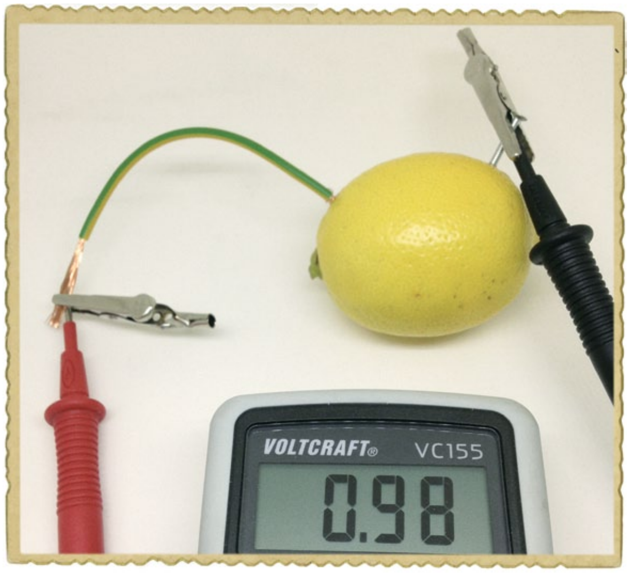


Step 2: Insert Electrodes into a Lemon

Roll and squeeze a lemon so that you break up the small juice packets inside it, but not enough to break the skin. Then, use a nail to make one hole in one end, push a copper wire into that hole, and push the nail into the other end, as shown. This is the first lemon battery!



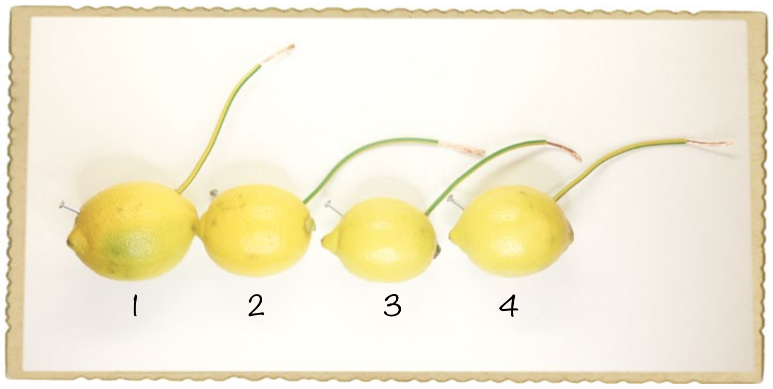
Get your multimeter, set it for DC voltage measurement, and test your lemon battery now. Place the positive test lead on the copper wire and the negative test lead on the nail. If everything works correctly, you should see a voltage of around 1 V on your multimeter.



Step 3: Create Four Lemon Batteries

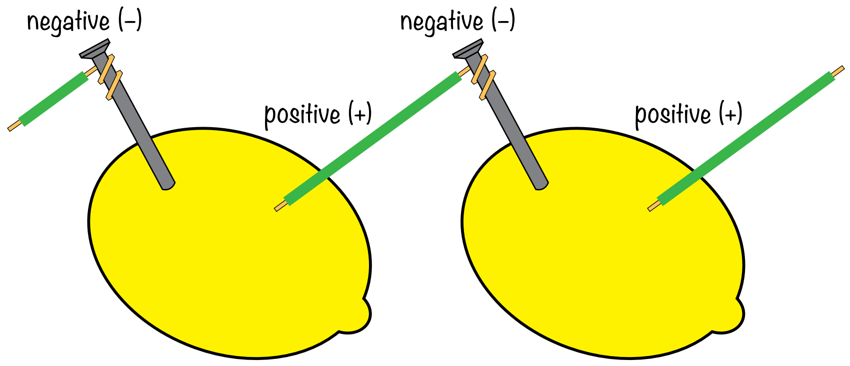
Even if you get 1 V out of your lemon, that's not enough to light an LED. Let's create several lemon batteries so we can get more electricity!

Just repeat the process described in Step 2 for the other lemons; each will become a battery. (If you don't have four lemons to spare, you can cut one lemon into four pieces.) Now you should have four lemon batteries.



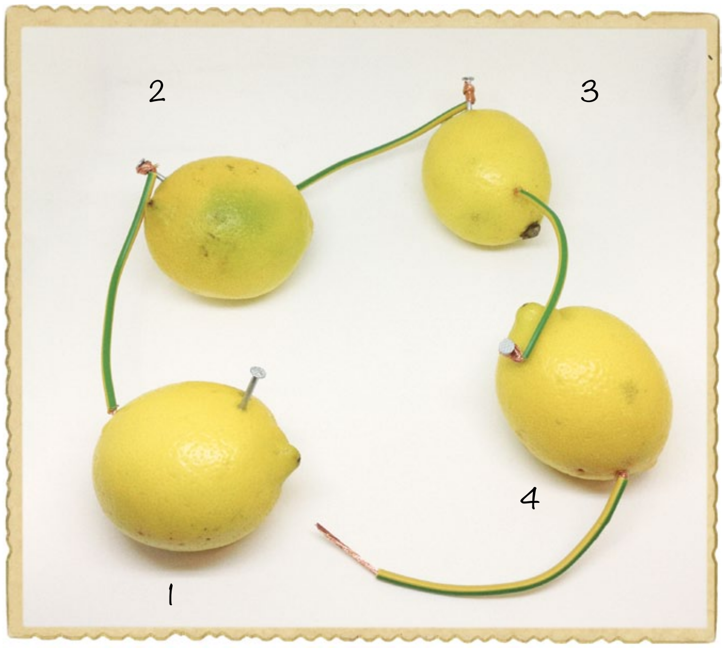
Step 4: Connect the Lemons in Series

To get a higher voltage with your lemon batteries, you'll need to connect them in series. To connect two lemons in series, you just connect the positive side of one lemon to the negative side of another. Remember, the copper wire is positive, and the nail is negative.

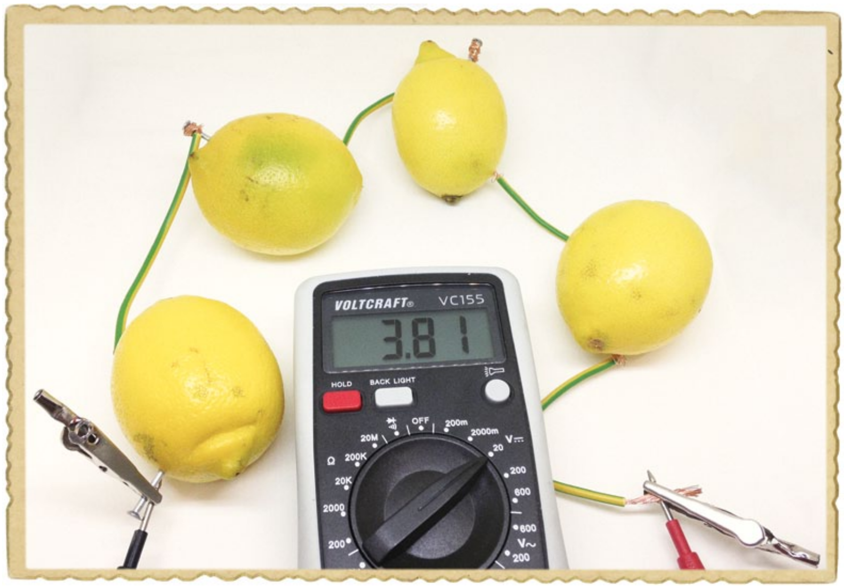


To wire four lemons in series, just repeat that process a couple more times. Line your lemons up in a row with the copper wires pointing to the right and number the lemons from 1 to 4, beginning from the left. Connect the copper wire from lemon 1 to the nail in lemon 2. Twist the wire onto the nail so that the metals connect without coming apart.

Connect the copper wire of lemon 2 to the nail in lemon 3, and connect the copper wire from lemon 3 to the nail in lemon 4. This should give you a row of four lemons, with an unconnected nail on lemon 1 and an unconnected copper wire on lemon 4. These are the positive and negative terminals for your big lemon battery, respectively.

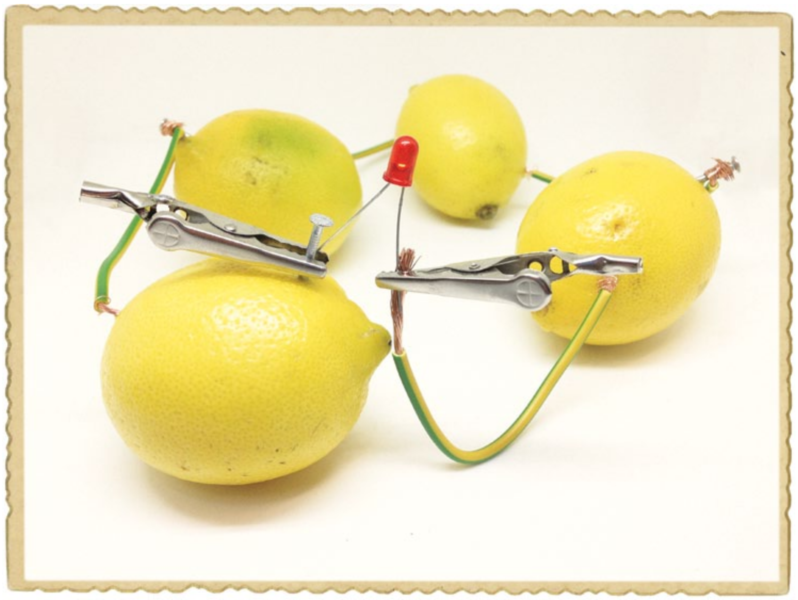


When you connect batteries in series, you can add their voltages to find your total. Four 1 V lemon batteries should give you 4 V. If you have a multimeter, measure the voltage between the two ends to see whether everything is connected. You should get a voltage of around 3.5 to 4 V.



Step 5: Test Your Lemon Battery

Let's connect the LED to the lemons! Connect the long leg from the LED to the copper wire, and connect the short leg to the nail, as shown. The LED should now light up.



Lemons aren't super powerful batteries (you'd never see anyone with a lemon connected to their computer, for example), so your LED will probably be very dim. After you finish building your lemon-powered circuit, turn off the light in your room, and you should see the LED glow.

Remember, when you're finished with your lemon battery, throw the lemons away—don't eat them!

TRY IT OUT: MORE FOOD BATTERIES!

When you're done making lemon batteries, test to see whether you can make batteries out of other fruits or vegetables. For example, what about a potato battery? Are you able to get more voltage, or is it the same as the voltage from the lemon?

Step 6: What If Your Lemon Light Doesn't Work?

If you can't see light from your LED, even in a dark room, check to see whether your LED is connected the right way. The long leg should be connected to the positive side of the battery, which is the copper wire.

Make sure the lemons are connected to each other only through the wires and nails. For example, if your lemons are sitting in a puddle of lemon juice, they could be connected through that. Just dry them off and move them somewhere else. Next, check that the copper wires are properly connected to the nails and that the nails and copper wires are actually touching the juice inside the lemons. Also, check that the nails and copper wires are not touching each other inside any of the lemons.

If the circuit still doesn't work, disconnect all the lemon batteries from each other. Then, use a multimeter to check that each lemon battery has some voltage. Connect two lemons in series, and check that you see a higher voltage. Connect the third lemon, and check that the voltage has increased again. Then, connect the fourth lemon and check that you have even more voltage.

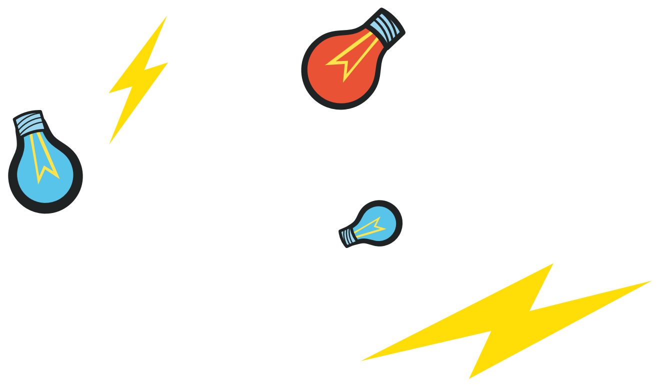
If you see a voltage but the LED doesn't light, then you probably just need some more power. Get another lemon or two, create some more batteries, and connect them in series with the rest.

WHAT'S NEXT?

In this chapter, you learned how to create your own electricity from magnetism and chemical reactions. You made your own shake generator, and you built a lemon battery to power an LED.

If you want to explore generators even more, I suggest trying to find a *dynamo* from an old bike. Unlike the generator you built in this chapter, a dynamo is a generator that gives you a DC voltage, like a battery, and dynamos are commonly used to power headlights on bikes. Cut some windmill blades out of some stiff cardboard or plastic, connect them to the dynamo, and see whether you can harvest energy from the wind.

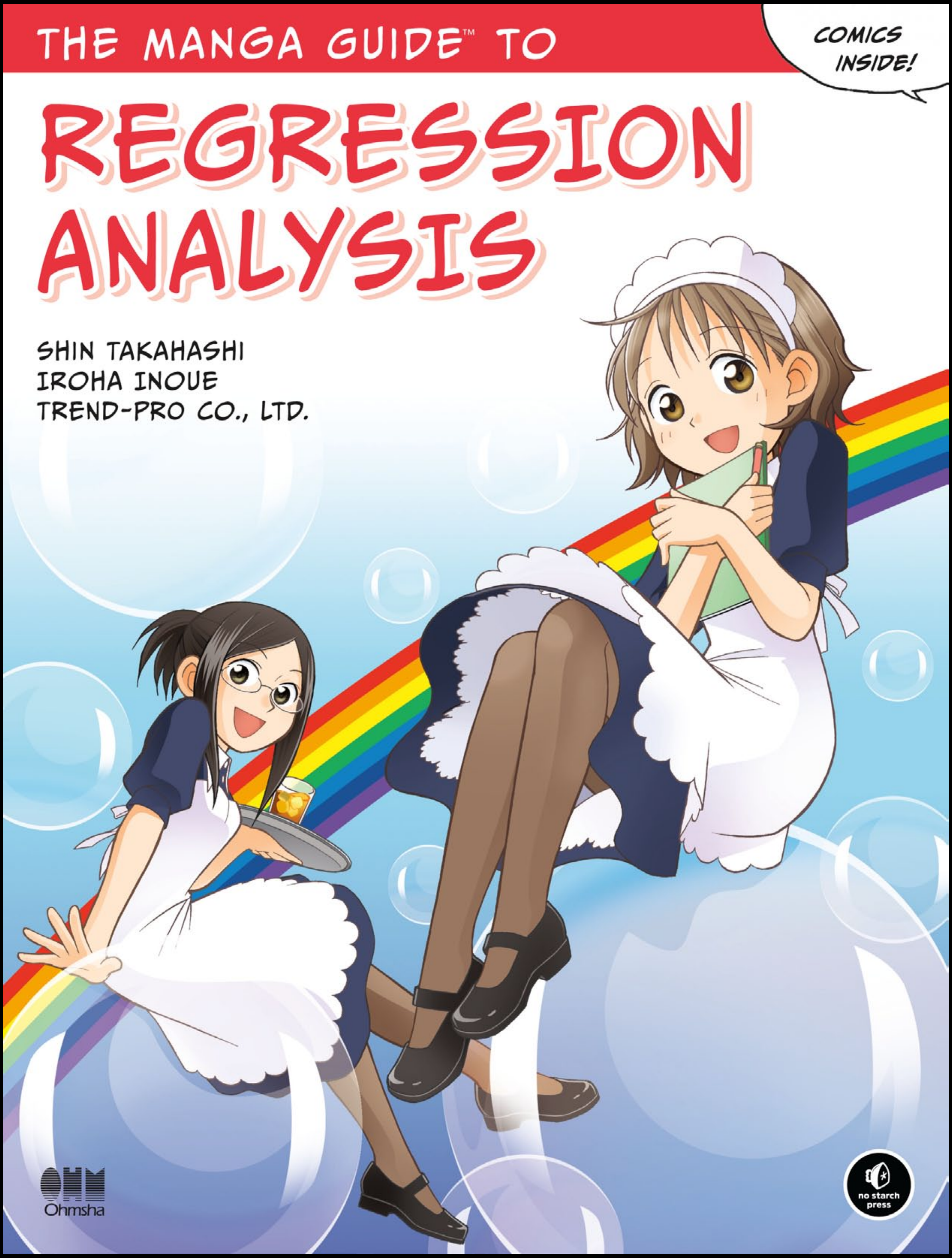
You've now met a few electronic components, including switches, LEDs, and motors. In the following chapters, you'll learn about even more components and graduate to building some real electronic circuits, like lights that blink, a touch-sensitive switch, and even your own electronic musical instrument!



THE MANGA GUIDE™ TO

REGRESSION ANALYSIS

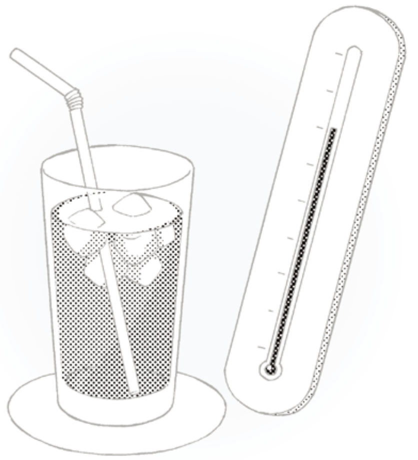
SHIN TAKAHASHI
IROHA INOUE
TREND-PRO CO., LTD.



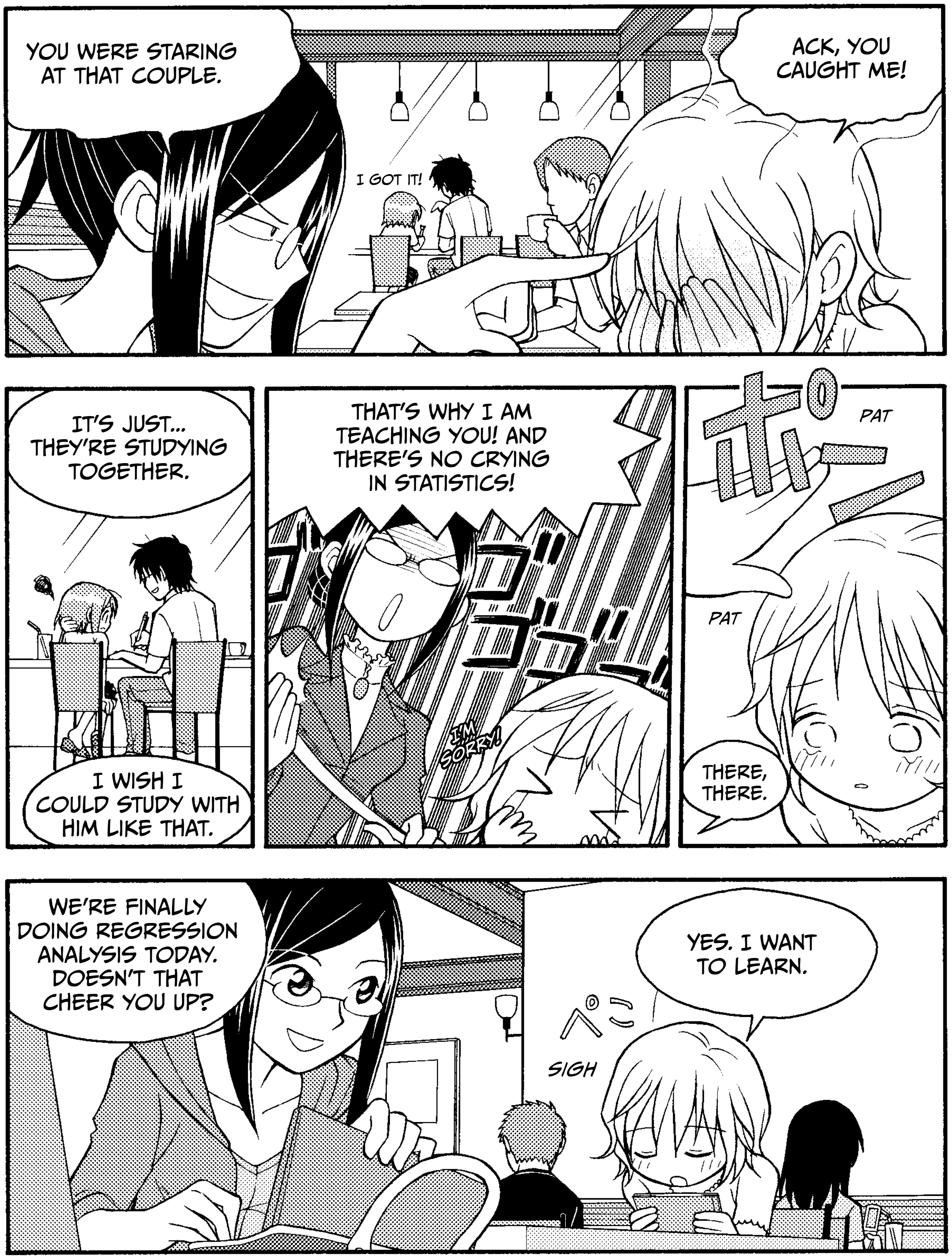
Ohmsha

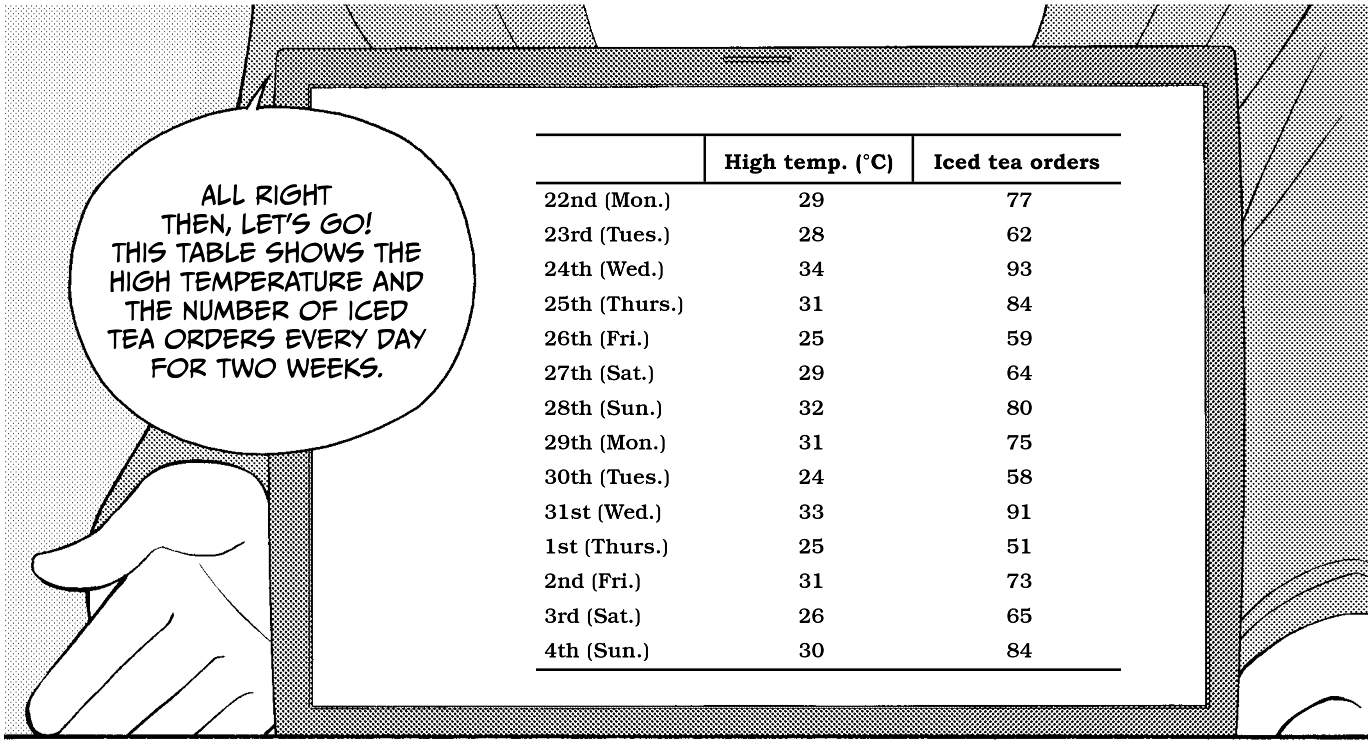
no starch press

2
SIMPLE REGRESSION ANALYSIS



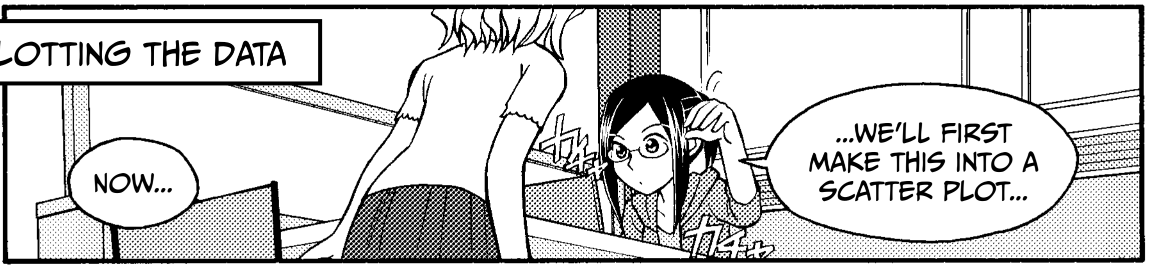
FIRST STEPS
THAT MEANS...
THERE IS A CONNECTION BETWEEN THE TWO, RIGHT?
EXACTLY!
WHERE DID YOU LEARN SO MUCH ABOUT REGRESSION ANALYSIS, MIU?
ぽ
MIU!
はっ
BLINK BLINK
EARTH TO MIU! ARE YOU THERE?

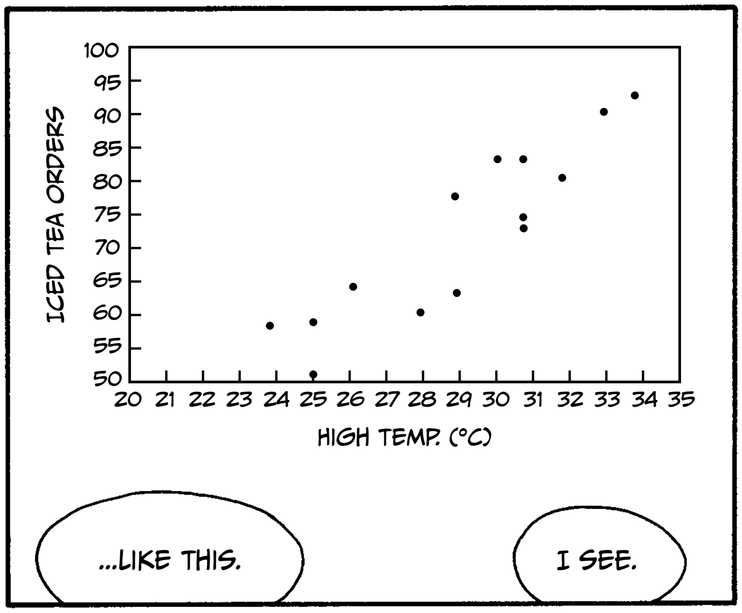

ACK, YOU CAUGHT ME!
YOU WERE STARING AT THAT COUPLE.
I GOT IT!
PAT
PAT
THERE, THERE.
THAT'S WHY I AM TEACHING YOU! AND THERE'S NO CRYING IN STATISTICS!
I'M SORRY!
IT'S JUST... THEY'RE STUDYING TOGETHER.
I WISH I COULD STUDY WITH HIM LIKE THAT.
YES. I WANT TO LEARN.
ぺこ
SIGH
WE'RE FINALLY DOING REGRESSION ANALYSIS TODAY. DOESN'T THAT CHEER YOU UP?


ALL RIGHT THEN, LET'S GO! THIS TABLE SHOWS THE HIGH TEMPERATURE AND THE NUMBER OF ICED TEA ORDERS EVERY DAY FOR TWO WEEKS.

	High temp. (°C)	Iced tea orders
22nd (Mon.)	29	77
23rd (Tues.)	28	62
24th (Wed.)	34	93
25th (Thurs.)	31	84
26th (Fri.)	25	59
27th (Sat.)	29	64
28th (Sun.)	32	80
29th (Mon.)	31	75
30th (Tues.)	24	58
31st (Wed.)	33	91
1st (Thurs.)	25	51
2nd (Fri.)	31	73
3rd (Sat.)	26	65
4th (Sun.)	30	84

PLOTTING THE DATA



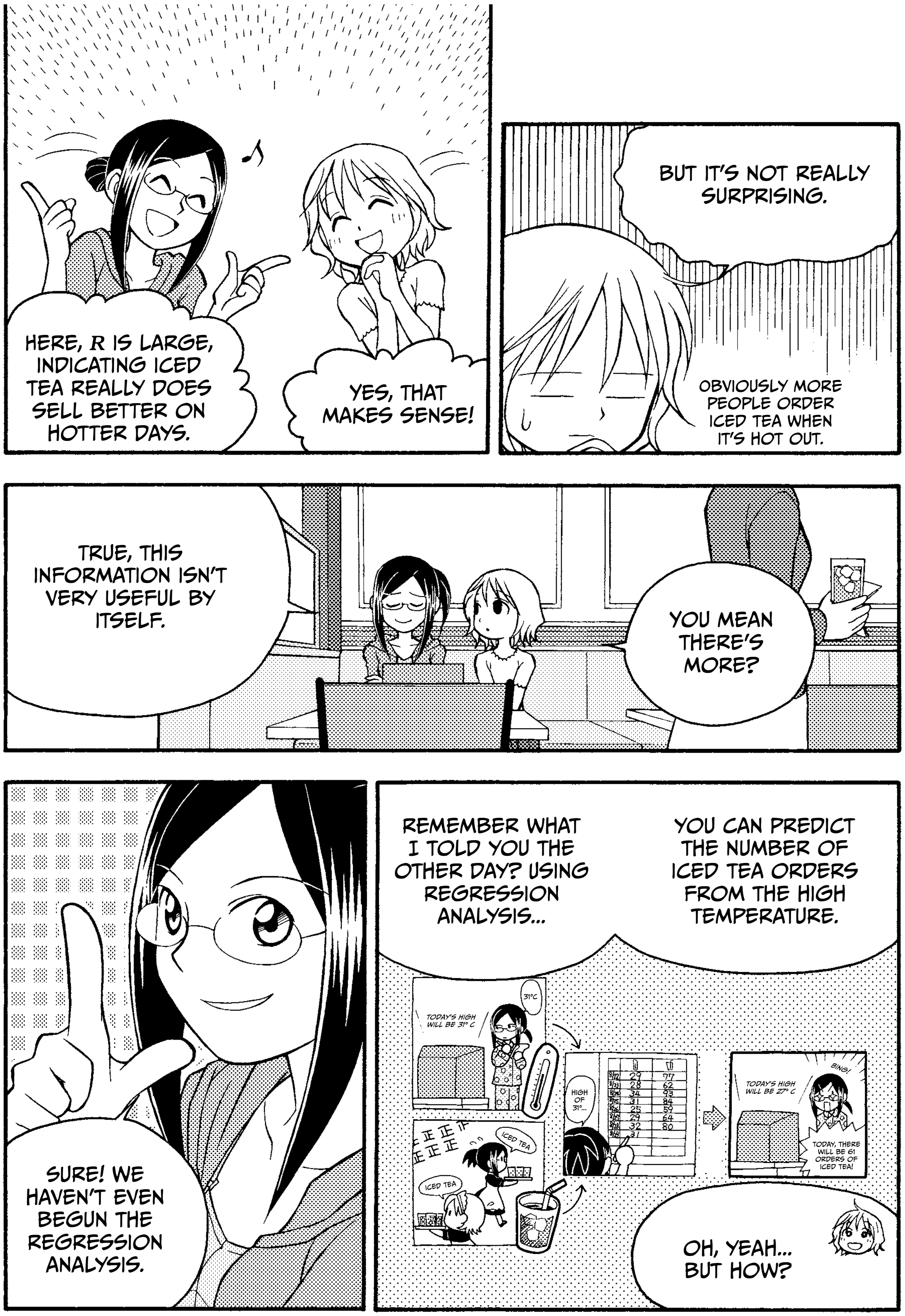


SEE HOW THE DOTS ROUGHLY LINE UP? THAT SUGGESTS THESE VARIABLES ARE CORRELATED. THE CORRELATION COEFFICIENT, CALLED *R*, INDICATES HOW STRONG THE CORRELATION IS.

$$R = 0.9069$$

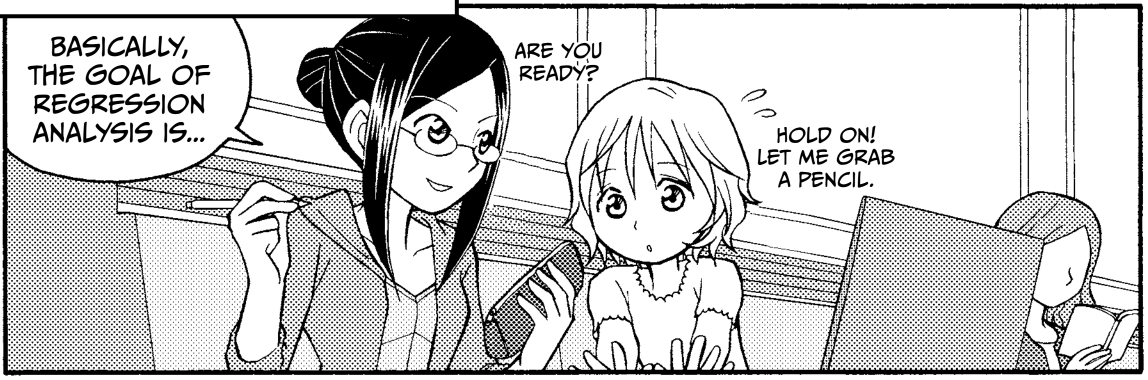
R RANGES FROM +1 TO −1, AND THE FURTHER IT IS FROM ZERO, THE STRONGER THE CORRELATION.* I'LL SHOW YOU HOW TO WORK OUT THE CORRELATION COEFFICIENT ON PAGE 78.

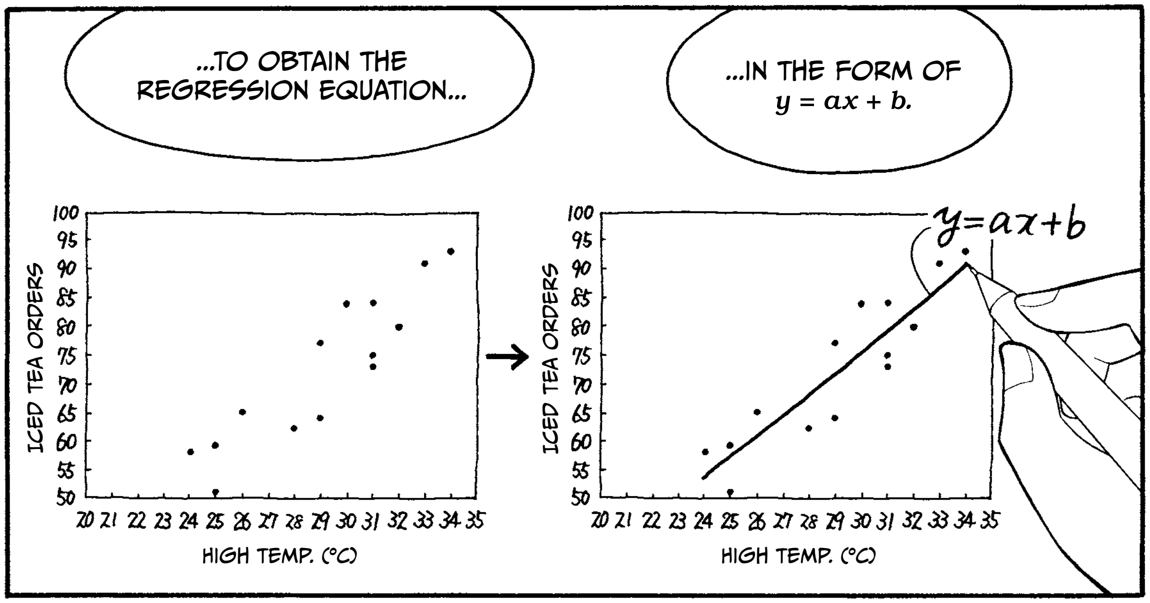
* A POSITIVE *R* VALUE INDICATES A POSITIVE RELATIONSHIP, MEANING AS x INCREASES, SO DOES y. A NEGATIVE *R* VALUE MEANS AS THE x VALUE INCREASES, THE y VALUE DECREASES.

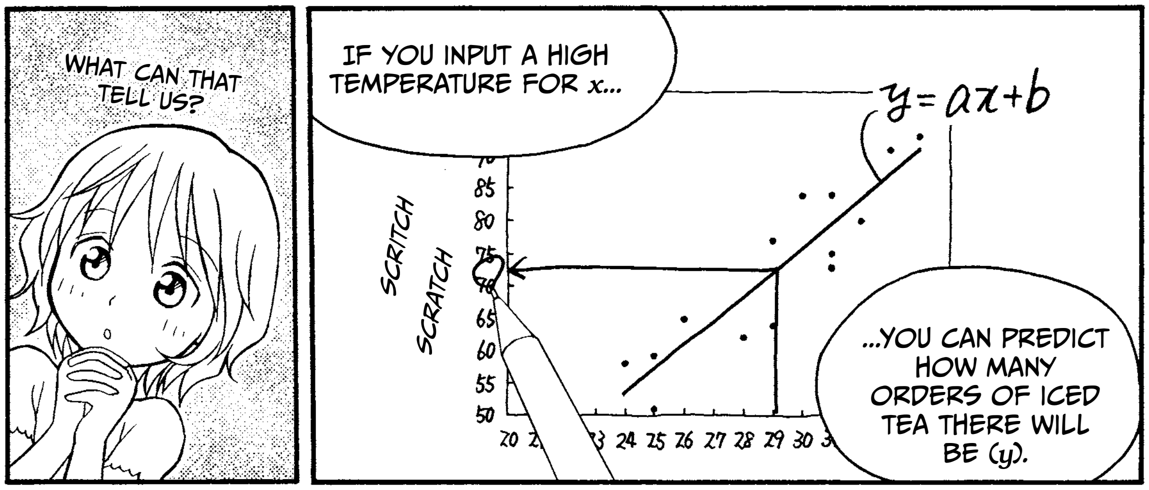

HERE, R IS LARGE, INDICATING ICED TEA REALLY DOES SELL BETTER ON HOTTER DAYS.
YES, THAT MAKES SENSE!
BUT IT'S NOT REALLY SURPRISING.
OBVIOUSLY MORE PEOPLE ORDER ICED TEA WHEN IT'S HOT OUT.
TRUE, THIS INFORMATION ISN'T VERY USEFUL BY ITSELF.
YOU MEAN THERE'S MORE?
SURE! WE HAVEN'T EVEN BEGUN THE REGRESSION ANALYSIS.
REMEMBER WHAT I TOLD YOU THE OTHER DAY? USING REGRESSION ANALYSIS...
YOU CAN PREDICT THE NUMBER OF ICED TEA ORDERS FROM THE HIGH TEMPERATURE.
TODAY'S HIGH WILL BE 31° C
31°C
ICED TEA
ICED TEA
HIGH OF 31°...
TODAY'S HIGH WILL BE 27° C
BING!
TODAY, THERE WILL BE 61 ORDERS OF ICED TEA!
OH, YEAH... BUT HOW?

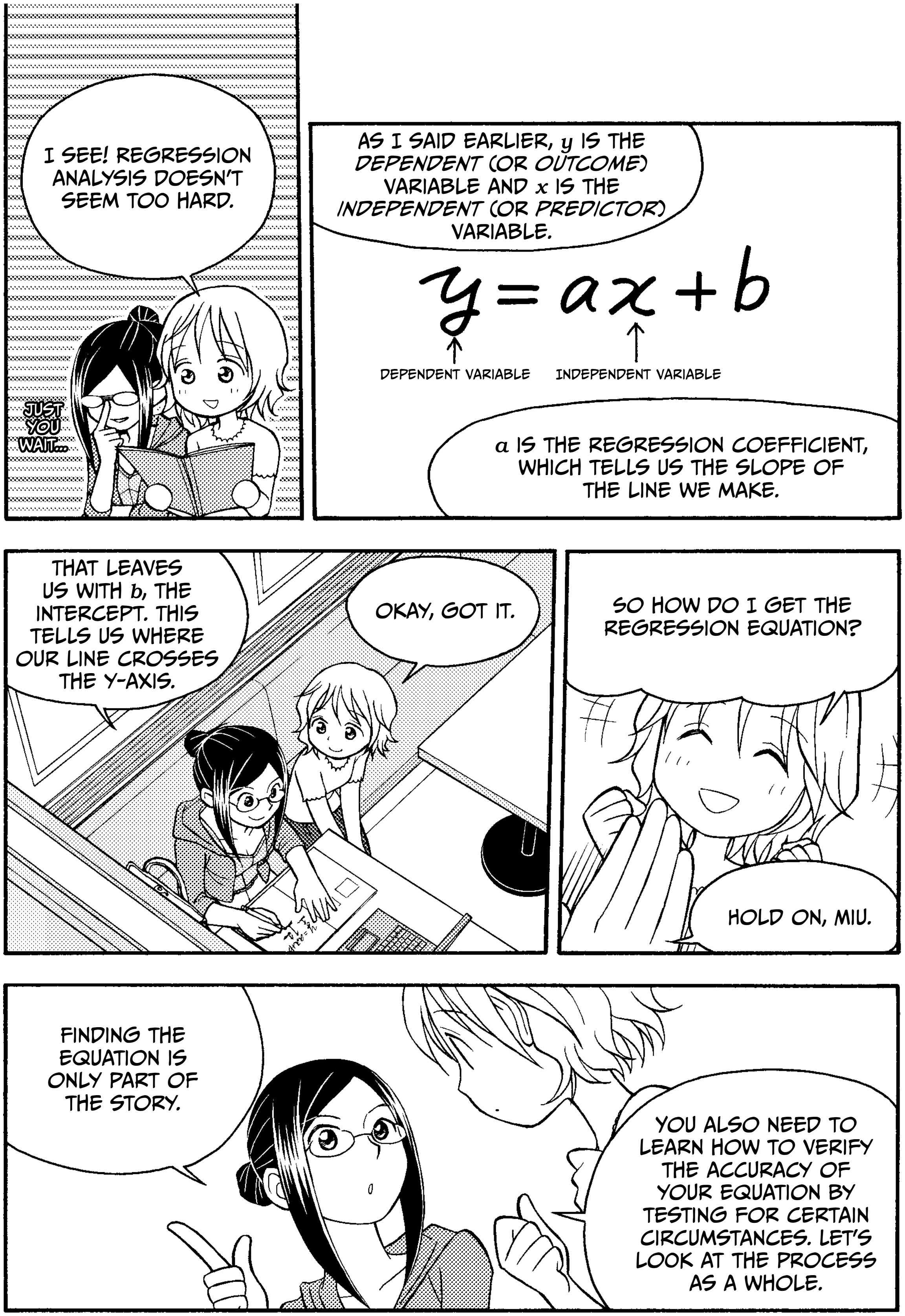
8/22	29	77
8/23	28	62
8/24	34	93
8/25	31	84
8/26	25	59
8/27	29	64
8/28	32	80
8/29	31	

THE REGRESSION EQUATION

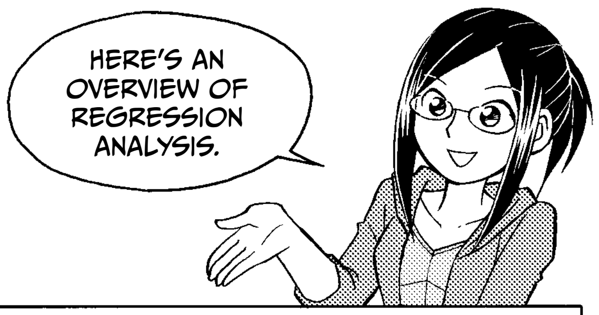

BASICALLY, THE GOAL OF REGRESSION ANALYSIS IS...
ARE YOU READY?
HOLD ON! LET ME GRAB A PENCIL.




WHAT CAN THAT TELL US?
IF YOU INPUT A HIGH TEMPERATURE FOR x...
$y=ax+b$
SCRITCH SCRATCH
...YOU CAN PREDICT HOW MANY ORDERS OF ICED TEA THERE WILL BE (y).

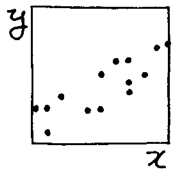

I SEE! REGRESSION ANALYSIS DOESN'T SEEM TOO HARD.
JUST YOU WAIT...
AS I SAID EARLIER, y IS THE *DEPENDENT* (OR *OUTCOME*) VARIABLE AND x IS THE *INDEPENDENT* (OR *PREDICTOR*) VARIABLE.
$y = ax + b$
DEPENDENT VARIABLE
INDEPENDENT VARIABLE
a IS THE REGRESSION COEFFICIENT, WHICH TELLS US THE SLOPE OF THE LINE WE MAKE.
THAT LEAVES US WITH b, THE INTERCEPT. THIS TELLS US WHERE OUR LINE CROSSES THE Y-AXIS.
OKAY, GOT IT.
SO HOW DO I GET THE REGRESSION EQUATION?
HOLD ON, MIU.
FINDING THE EQUATION IS ONLY PART OF THE STORY.
YOU ALSO NEED TO LEARN HOW TO VERIFY THE ACCURACY OF YOUR EQUATION BY TESTING FOR CERTAIN CIRCUMSTANCES. LET'S LOOK AT THE PROCESS AS A WHOLE.

GENERAL REGRESSION ANALYSIS PROCEDURE



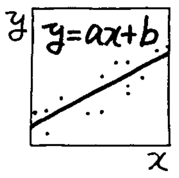
STEP 1

DRAW A SCATTER PLOT OF THE INDEPENDENT VARIABLE VERSUS THE DEPENDENT VARIABLE. IF THE DOTS LINE UP, THE VARIABLES MAY BE CORRELATED.



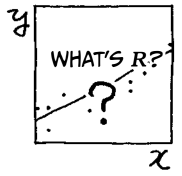
STEP 2

CALCULATE THE REGRESSION EQUATION.



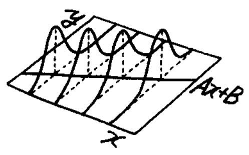
STEP 3

CALCULATE THE CORRELATION COEFFICIENT (*R*) AND ASSESS OUR POPULATION AND ASSUMPTIONS.



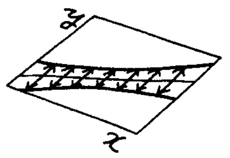
STEP 4

CONDUCT THE ANALYSIS OF VARIANCE.



STEP 5

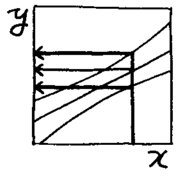
CALCULATE THE CONFIDENCE INTERVALS.

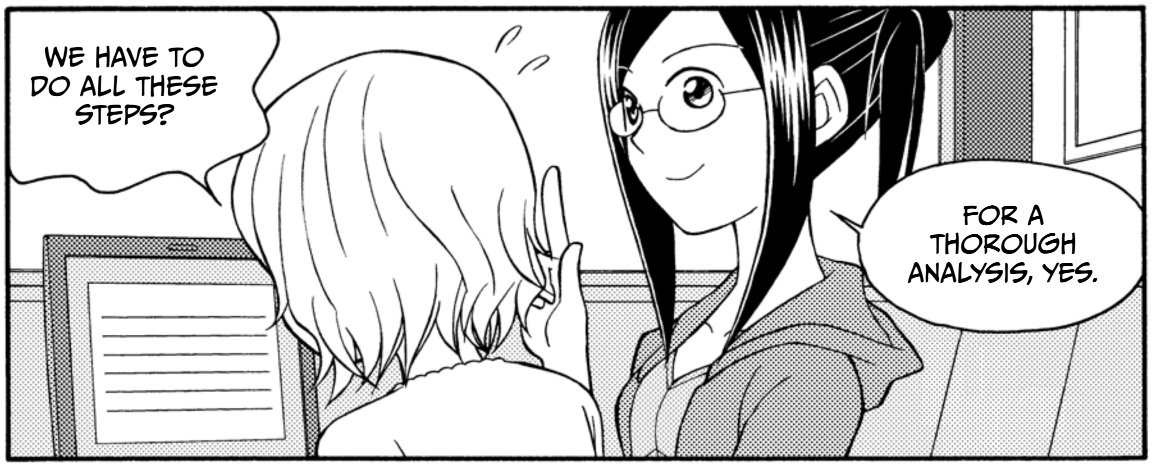


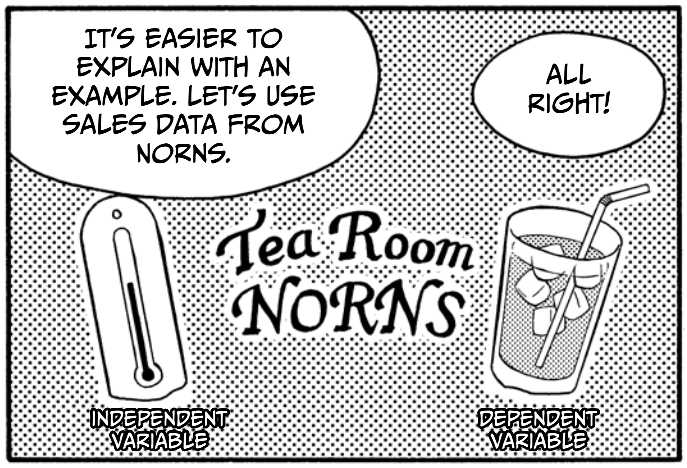
REGRESSION DIAGNOSTICS

STEP 6

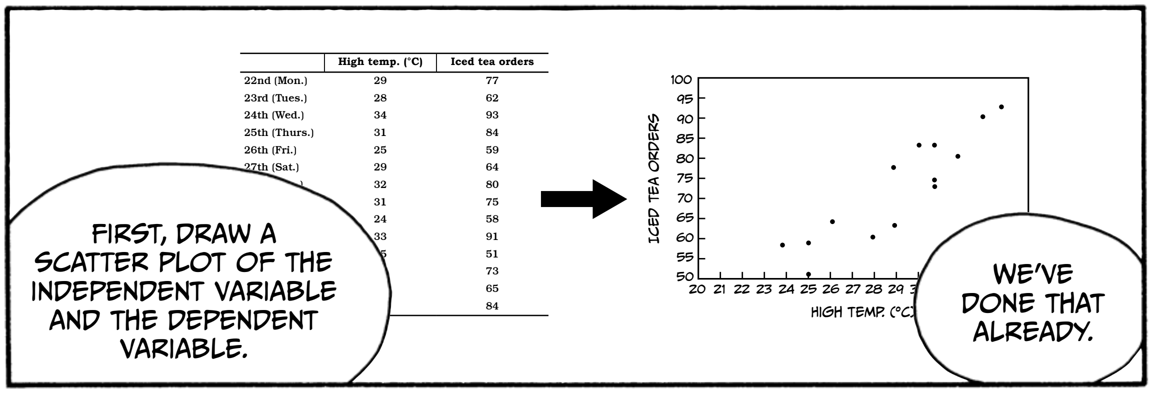
MAKE A PREDICTION!



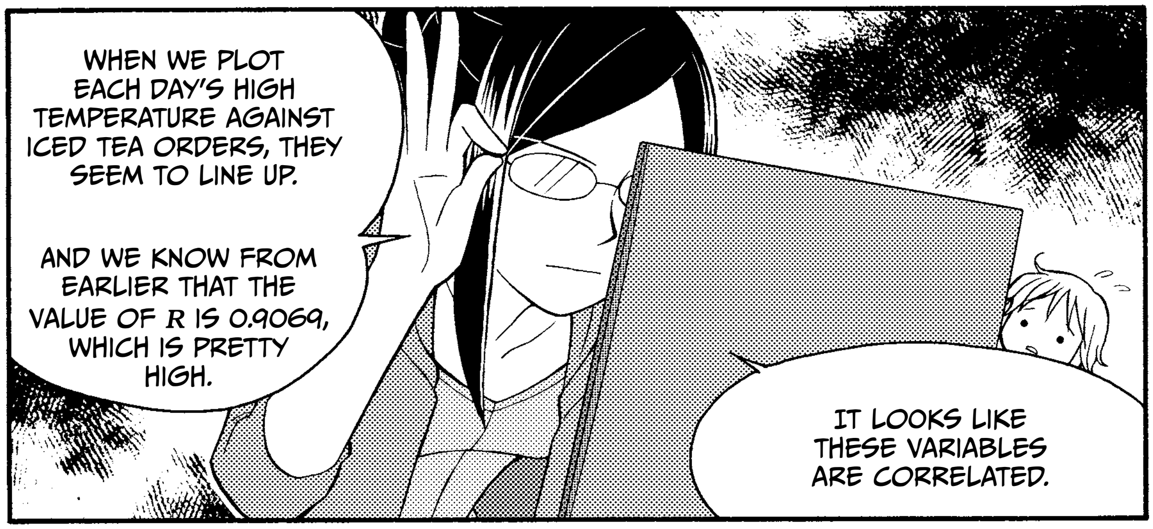


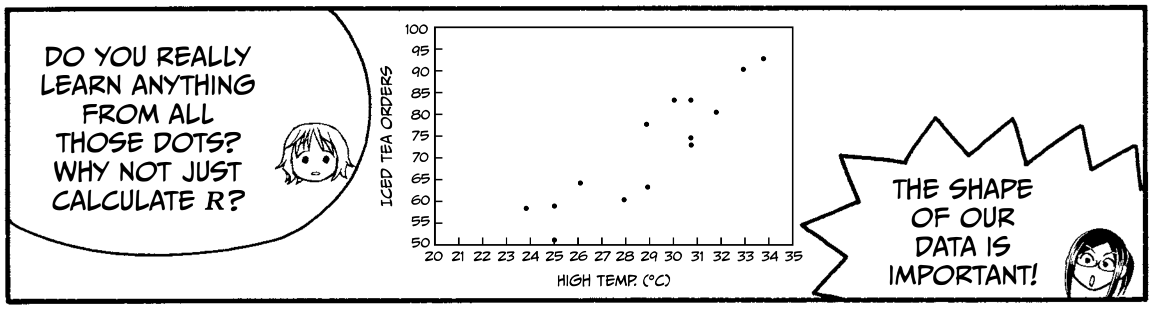

IT'S EASIER TO EXPLAIN WITH AN EXAMPLE. LET'S USE SALES DATA FROM NORNS.
ALL RIGHT!
Tea Room NORNS
INDEPENDENT VARIABLE
DEPENDENT VARIABLE

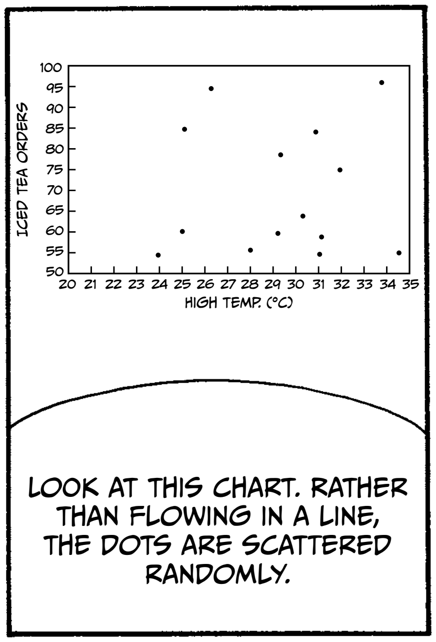
STEP 1: DRAW A SCATTER PLOT OF THE INDEPENDENT VARIABLE VERSUS THE DEPENDENT VARIABLE. IF THE DOTS LINE UP, THE VARIABLES MAY BE CORRELATED.

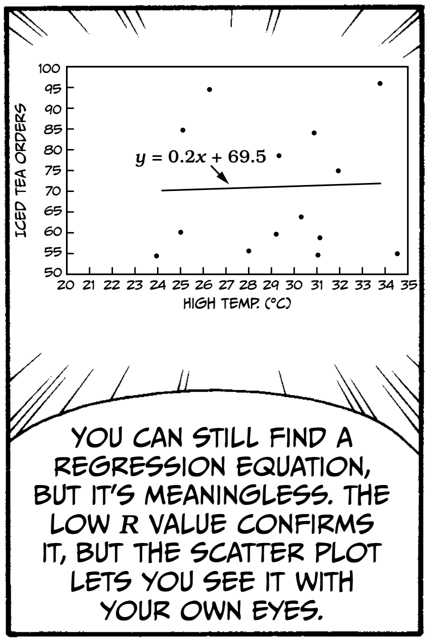


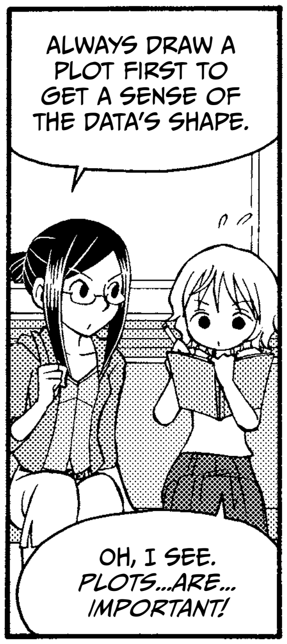
	High temp. (°C)	Iced tea orders
22nd (Mon.)	29	77
23rd (Tues.)	28	62
24th (Wed.)	34	93
25th (Thurs.)	31	84
26th (Fri.)	25	59
27th (Sat.)	29	64
[illegible]	32	80
[illegible]	31	75
[illegible]	24	58
[illegible]	33	91
[illegible]	5	51
[illegible]	[illegible]	73
[illegible]	[illegible]	65
[illegible]	[illegible]	84


WHEN WE PLOT EACH DAY'S HIGH TEMPERATURE AGAINST ICED TEA ORDERS, THEY SEEM TO LINE UP.
AND WE KNOW FROM EARLIER THAT THE VALUE OF *R* IS 0.9069, WHICH IS PRETTY HIGH.
IT LOOKS LIKE THESE VARIABLES ARE CORRELATED.

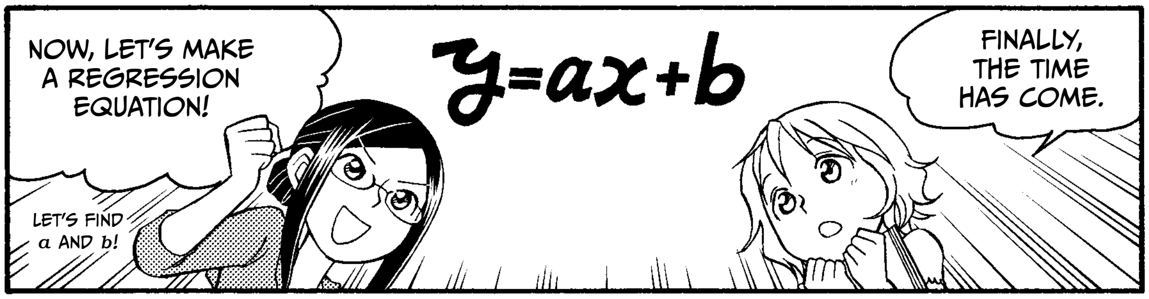

DO YOU REALLY LEARN ANYTHING FROM ALL THOSE DOTS? WHY NOT JUST CALCULATE *R*?
ICED TEA ORDERS
HIGH TEMP. (°C)
THE SHAPE OF OUR DATA IS IMPORTANT!

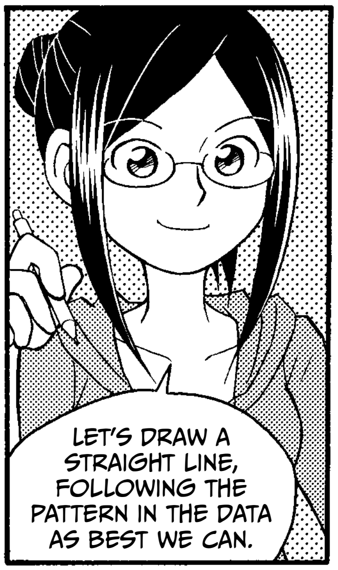

ICED TEA ORDERS
HIGH TEMP. (°C)
LOOK AT THIS CHART. RATHER THAN FLOWING IN A LINE, THE DOTS ARE SCATTERED RANDOMLY.

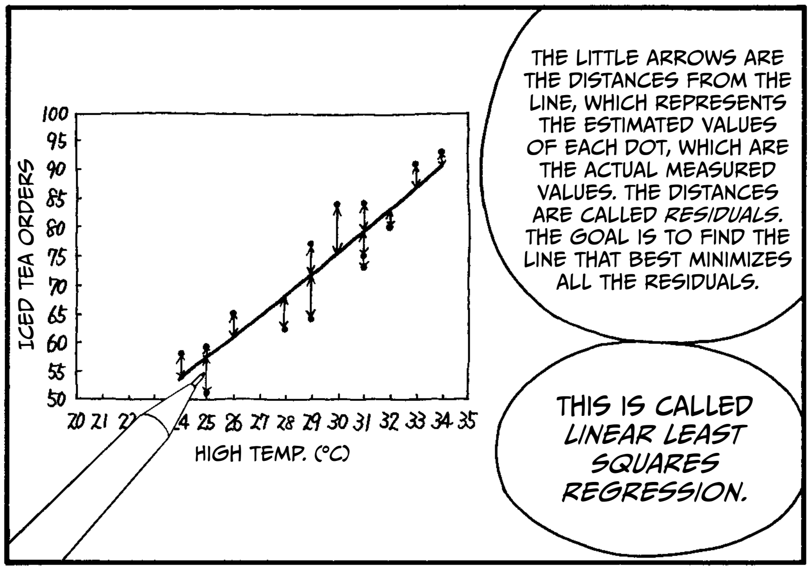

ICED TEA ORDERS
$y = 0.2x + 69.5$
HIGH TEMP. (°C)
YOU CAN STILL FIND A REGRESSION EQUATION, BUT IT'S MEANINGLESS. THE LOW *R* VALUE CONFIRMS IT, BUT THE SCATTER PLOT LETS YOU SEE IT WITH YOUR OWN EYES.

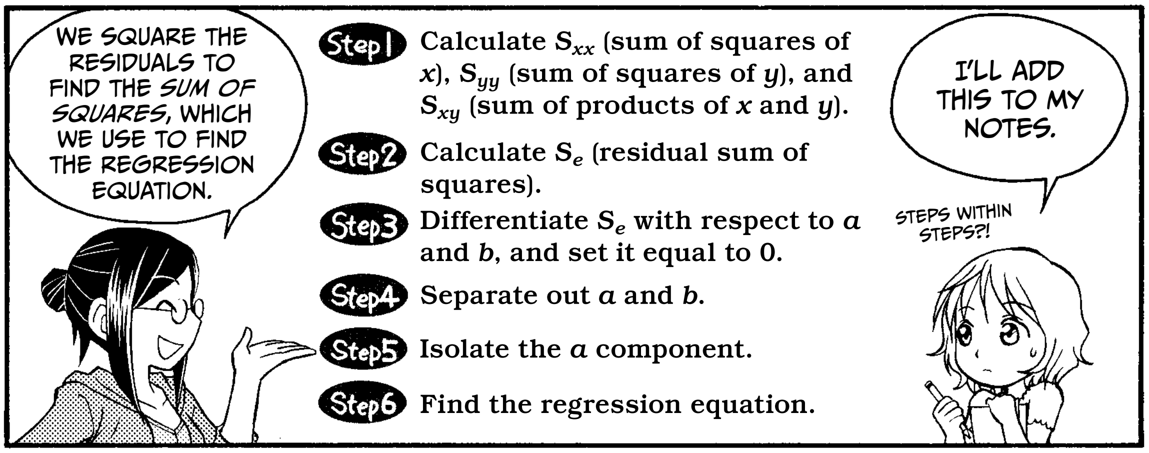

ALWAYS DRAW A PLOT FIRST TO GET A SENSE OF THE DATA'S SHAPE.
OH, I SEE. *PLOTS...ARE... IMPORTANT!*

STEP 2: CALCULATE THE REGRESSION EQUATION.


NOW, LET'S MAKE A REGRESSION EQUATION!
LET'S FIND *a* AND *b*!
$y = ax + b$
FINALLY, THE TIME HAS COME.


LET'S DRAW A STRAIGHT LINE, FOLLOWING THE PATTERN IN THE DATA AS BEST WE CAN.




WE SQUARE THE RESIDUALS TO FIND THE *SUM OF SQUARES*, WHICH WE USE TO FIND THE REGRESSION EQUATION.
I'LL ADD THIS TO MY NOTES.
STEPS WITHIN STEPS?!

Step 1 Calculate S_{xx} (sum of squares of x), S_{yy} (sum of squares of y), and S_{xy} (sum of products of x and y).

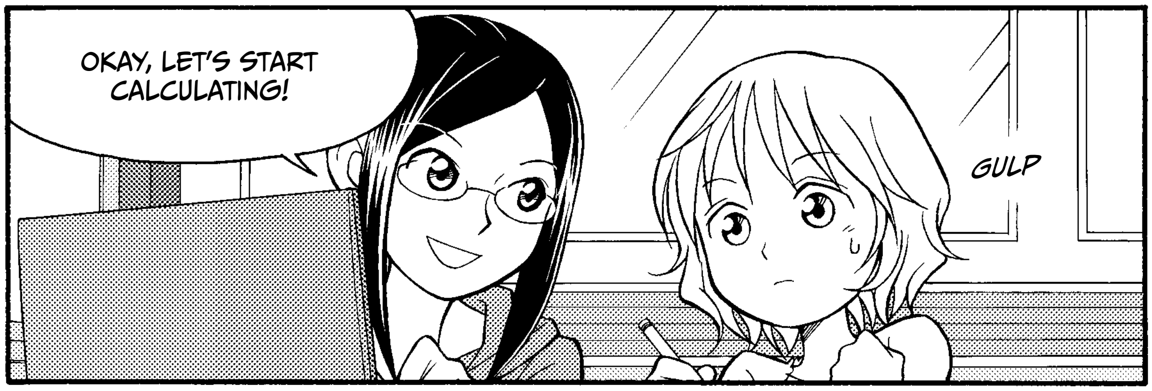
Step 2 Calculate S_e (residual sum of squares).

Step 3 Differentiate S_e with respect to a and b, and set it equal to 0.

Step 4 Separate out a and b.

Step 5 Isolate the a component.

Step 6 Find the regression equation.



Step 1 Find

- The sum of squares of x, S_{xx}: $(x-\bar{x})^2$
- The sum of squares of y, S_{yy}: $(y-\bar{y})^2$
- The sum of products of x and y, S_{xy}: $(x-\bar{x})(y-\bar{y})$

Note: The bar over a variable (like $\bar{x}$) is a notation that means *average*. We can call this variable x-bar.

	High temp. in °C x	Iced tea orders y	$x-\bar{x}$	$y-\bar{y}$	$(x-\bar{x})^2$	$(y-\bar{y})^2$	$(x-\bar{x})(y-\bar{y})$
22nd (Mon.)	29	77	−0.1	4.4	0.0	19.6	−0.6
23rd (Tues.)	28	62	−1.1	−10.6	1.3	111.8	12.1
24th (Wed.)	34	93	4.9	20.4	23.6	417.3	99.2
25th (Thurs.)	31	84	1.9	11.4	3.4	130.6	21.2
26th (Fri.)	25	59	−4.1	−13.6	17.2	184.2	56.2
27th (Sat.)	29	64	−0.1	−8.6	0.0	73.5	1.2
28th (Sun.)	32	80	2.9	7.4	8.2	55.2	21.2
29th (Mon.)	31	75	1.9	2.4	3.4	5.9	4.5
30th (Tues.)	24	58	−5.1	−14.6	26.4	212.3	74.9
31st (Wed.)	33	91	3.9	18.4	14.9	339.6	71.1
1st (Thurs.)	25	51	−4.1	−21.6	17.2	465.3	89.4
2nd (Fri.)	31	73	1.9	0.4	3.4	0.2	0.8
3rd (Sat.)	26	65	−3.1	−7.6	9.9	57.8	23.8
4th (Sun.)	30	84	0.9	11.4	0.7	130.6	9.8
Sum	408	1016	0	0	129.7 $\downarrow$ S_{xx}	2203.4 $\downarrow$ S_{yy}	484.9 $\downarrow$ S_{xy}
Average	29.1 $\downarrow$ $\bar{x}$	72.6 $\downarrow$ $\bar{y}$					

* SOME OF THE FIGURES IN THIS CHAPTER ARE ROUNDED FOR THE SAKE OF PRINTING, BUT CALCULATIONS ARE DONE USING THE FULL, UNROUNDED VALUES RESULTING FROM THE RAW DATA UNLESS OTHERWISE STATED.

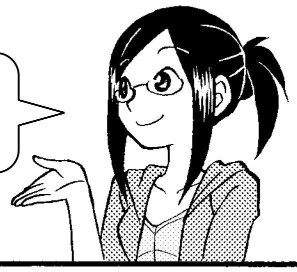
Step2 **Find the residual sum of squares, S_e.**

- **y is the observed value.**
- **$\hat{y}$ is the the estimated value based on our regression equation.**
- **$y - \hat{y}$ is called the residual and is written as e.**

Note: The caret in $\hat{y}$ is affectionately called a *hat*, so we call this parameter estimate y-hat.

	High temp. in °C x	**Actual iced tea orders** y	**Predicted iced tea orders** $\hat{y} = ax + b$	**Residuals (e)** $y - \hat{y}$	**Squared residuals** $(y - \hat{y})^2$
22nd (Mon.)	29	77	$a \times 29 + b$	$77 - (a \times 29 + b)$	$[77 - (a \times 29 + b)]^2$
23rd (Tues.)	28	62	$a \times 28 + b$	$62 - (a \times 28 + b)$	$[62 - (a \times 28 + b)]^2$
24th (Wed.)	34	93	$a \times 34 + b$	$93 - (a \times 34 + b)$	$[93 - (a \times 34 + b)]^2$
25th (Thurs.)	31	84	$a \times 31 + b$	$84 - (a \times 31 + b)$	$[84 - (a \times 31 + b)]^2$
26th (Fri.)	25	59	$a \times 25 + b$	$59 - (a \times 25 + b)$	$[59 - (a \times 25 + b)]^2$
27th (Sat.)	29	64	$a \times 29 + b$	$64 - (a \times 29 + b)$	$[64 - (a \times 29 + b)]^2$
28th (Sun.)	32	80	$a \times 32 + b$	$80 - (a \times 32 + b)$	$[80 - (a \times 32 + b)]^2$
29th (Mon.)	31	75	$a \times 31 + b$	$75 - (a \times 31 + b)$	$[75 - (a \times 31 + b)]^2$
30th (Tues.)	24	58	$a \times 24 + b$	$58 - (a \times 24 + b)$	$[58 - (a \times 24 + b)]^2$
31st (Wed.)	33	91	$a \times 33 + b$	$91 - (a \times 33 + b)$	$[91 - (a \times 33 + b)]^2$
1st (Thurs.)	25	51	$a \times 25 + b$	$51 - (a \times 25 + b)$	$[51 - (a \times 25 + b)]^2$
2nd (Fri.)	31	73	$a \times 31 + b$	$73 - (a \times 31 + b)$	$[73 - (a \times 31 + b)]^2$
3rd (Sat.)	26	65	$a \times 26 + b$	$65 - (a \times 26 + b)$	$[65 - (a \times 26 + b)]^2$
4th (Sun.)	30	84	$a \times 30 + b$	$84 - (a \times 30 + b)$	$[84 - (a \times 30 + b)]^2$
Sum	408	1016	$408a + 14b$	$1016 - (408a + 14b)$	S_e
Average	29.1	72.6	$29.1a + b$ $= \bar{x}a + b$	$72.6 - (29.1a + b)$ $= \bar{y} - (\bar{x}a + b)$	$= \frac{S_e}{14}$
	$\downarrow$ $\bar{x}$	$\downarrow$ $\bar{y}$			

$$S_e = \left[77 - (a \times 29 + b)\right]^2 + \cdots + \left[84 - (a \times 30 + b)\right]^2$$


THE SUM OF THE RESIDUALS SQUARED IS CALLED THE *RESIDUAL SUM OF SQUARES*. IT IS WRITTEN AS S_e OR RSS.

Step3 Differentiate S_e with respect to a and b, and set it equal to 0. When differentiating $y = (ax+b)^n$ with respect to x, the result is $\frac{dy}{dx} = n(ax+b)^{n-1} \times a$.

- Differentiate with respect to a.

$$\frac{dS_e}{da} = 2\left[77-(29a+b)\right]\times(-29)+\cdots+2\left[84-(30a+b)\right]\times(-30)=0 \quad ❶$$

- Differentiate with respect to b.

$$\frac{dS_e}{db} = 2\left[77-(29a+b)\right]\times(-1)+\cdots+2\left[84-(30a+b)\right]\times(-1)=0 \quad ❷$$

Step4 Rearrange ❶ and ❷ from the previous step.

Rearrange ❶.

$$2\left[77-(29a+b)\right]\times(-29)+\cdots+2\left[84-(30a+b)\right]\times(-30)=0$$

$$\left[77-(29a+b)\right]\times(-29)+\cdots+\left[84-(30a+b)\right]\times(-30)=0$$ DIVIDE BOTH SIDES BY 2.

$$29\left[(29a+b)-77\right]+\cdots+30\left[(30a+b)-84\right]=0$$ MULTIPLY BY -1.

$$(29\times29a+29\times b-29\times77)+\cdots+(30\times30a+30\times b-30\times84)=0$$ MULTIPLY.

❸ $$\left(29^2+\cdots+30^2\right)a+(29+\cdots+30)b-(29\times77+\cdots+30\times84)=0$$ SEPARATE OUT a AND b.

Rearrange ❷.

$$2\left[77-(29a+b)\right]\times(-1)+\cdots+2\left[84-(30a+b)\right]\times(-1)=0$$

$$\left[77-(29a+b)\right]\times(-1)+\cdots+\left[84-(30a+b)\right]\times(-1)=0$$ DIVIDE BOTH SIDES BY 2.

$$\left[(29a+b)-77\right]+\cdots+\left[(30a+b)-84\right]=0$$ MULTIPLY BY -1.

$$(29+\cdots+30)a+\underbrace{b+\cdots+b}_{14}-(77+\cdots+84)=0$$ SEPARATE OUT a AND b.

$$(29+\cdots+30)a+14b-(77+\cdots+84)=0$$

$$14b=(77+\cdots+84)-(29+\cdots+30)a$$ SUBTRACT $14b$ FROM BOTH SIDES AND MULTIPLY BY -1.

❹ $$b=\frac{77+\cdots+84}{14}-\frac{29+\cdots+30}{14}a$$ ISOLATE b ON THE LEFT SIDE OF THE EQUATION.

❺ $$b=\bar{y}-\bar{x}a$$ THE COMPONENTS IN ❹ ARE THE AVERAGES OF y AND x.

Step5 **Plug the value of *b* found in ❹ into line ❸ (❸ and ❹ are the results from Step 4).**

❸ $$\left(29^2+\cdots+30^2\right)a+\left(29+\cdots+30\right)\overset{❹}{\left(\frac{77+\cdots+84}{14}-\frac{29+\cdots+30}{14}a\right)}-\left(29\times77+\cdots+30\times84\right)=0$$

NOW *a* IS THE ONLY VARIABLE.

$$\left(29^2+\cdots+30^2\right)a+\frac{\left(29+\cdots+30\right)\left(77+\cdots+84\right)}{14}-\frac{\left(29+\cdots+30\right)^2}{14}a-\left(29\times77+\cdots+30\times84\right)=0$$

$$\left[\left(29^2+\cdots+30^2\right)-\frac{\left(29+\cdots+30\right)^2}{14}\right]a+\frac{\left(29+\cdots+30\right)\left(77+\cdots+84\right)}{14}-\left(29\times77+\cdots+30\times84\right)=0$$

COMBINE THE *a* TERMS.

$$\left[\left(29^2+\cdots+30^2\right)-\frac{\left(29+\cdots+30\right)^2}{14}\right]a=\left(29\times77+\cdots+30\times84\right)-\frac{\left(29+\cdots+30\right)\left(77+\cdots+84\right)}{14}$$

TRANSPOSE.

Rearrange the left side of the equation.

$$\left(29^2+\cdots+30^2\right)-\frac{\left(29+\cdots+30\right)^2}{14}$$

$$=\left(29^2+\cdots+30^2\right)-2\times\frac{\left(29+\cdots+30\right)^2}{14}+\frac{\left(29+\cdots+30\right)^2}{14}$$

WE ADD AND SUBTRACT $\frac{\left(29+\cdots+30\right)^2}{14}$.

$$=\left(29^2+\cdots+30^2\right)-2\times\left(29+\cdots+30\right)\times\frac{29+\cdots+30}{14}+\left(\frac{29+\cdots+30}{14}\right)^2\times14$$

THE LAST TERM IS MULTIPLIED BY $\frac{14}{14}$.

$$=\left(29^2+\cdots+30^2\right)-2\times\left(29+\cdots+30\right)\times\bar{x}+\left(\bar{x}\right)^2\times14$$

$\bar{x}=\frac{29+\cdots+30}{14}$

$$=\left(29^2+\cdots+30^2\right)-2\times\left(29+\cdots+30\right)\times\bar{x}+\underbrace{\left(\bar{x}\right)^2+\cdots+\left(\bar{x}\right)^2}_{14}$$

$$=\left[29^2-2\times29\times\bar{x}+\left(\bar{x}\right)^2\right]+\cdots+\left[30^2-2\times30\times\bar{x}+\left(\bar{x}\right)^2\right]$$

$$=\left(29-\bar{x}\right)^2+\cdots+\left(30-\bar{x}\right)^2$$

$$=S_{xx}$$

Rearrange the right side of the equation.

$$\left(29\times77+\cdots+30\times84\right)-\frac{\left(29+\cdots+30\right)\left(77+\cdots+84\right)}{14}$$

$$=\left(29\times77+\cdots+30\times84\right)-\frac{29+\cdots+30}{14}\times\frac{77+\cdots+84}{14}\times14$$

$$=\left(29\times77+\cdots+30\times84\right)-\bar{x}\times\bar{y}\times14$$

$$=\left(29\times77+\cdots+30\times84\right)-\bar{x}\times\bar{y}\times14-\bar{x}\times\bar{y}\times14+\bar{x}\times\bar{y}\times14$$

WE ADD AND SUBTRACT $\bar{x}\times\bar{y}\times14$.

$$=\left(29\times77+\cdots+30\times84\right)-\frac{29+\cdots+30}{14}\times\bar{y}\times14-\bar{x}\times\frac{77+\cdots+84}{14}\times14+\bar{x}\times\bar{y}\times14$$

$$=\left(29\times77+\cdots+30\times84\right)-\left(29+\cdots+30\right)\bar{y}-\bar{x}\left(77+\cdots+84\right)+\bar{x}\times\bar{y}\times14$$

$$=\left(29\times77+\cdots+30\times84\right)-\left(29+\cdots+30\right)\bar{y}-\left(77+\cdots+84\right)\bar{x}+\underbrace{\bar{x}\times\bar{y}+\cdots+\bar{x}\times\bar{y}}_{14}$$

$$=\left(29-\bar{x}\right)\left(77-\bar{y}\right)+\cdots+\left(30-\bar{x}\right)\left(84-\bar{y}\right)$$

$$=S_{xy}$$

$$S_{xx}a=S_{xy}$$

❻ $$a=\frac{S_{xy}}{S_{xx}}$$

ISOLATE *a* ON THE LEFT SIDE OF THE EQUATION.

Step 6 Calculate the regression equation.

From ❻ in Step 5, $a = \frac{S_{xy}}{S_{xx}}$. From ❺ in Step 4, $b = \bar{y} - \bar{x}a$.

If we plug in the values we calculated in Step 1,

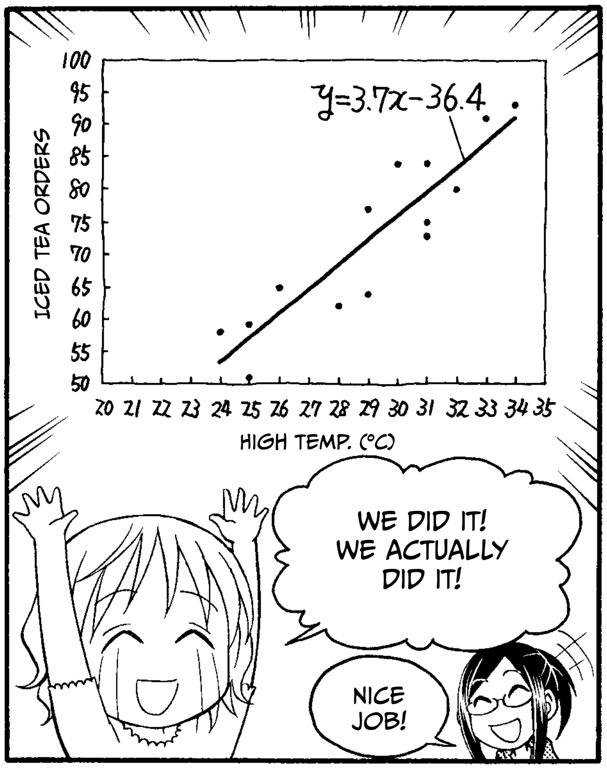
$$\begin{cases} a = \dfrac{S_{xx}}{S_{xy}} = \dfrac{484.9}{129.7} = 3.7 \\ b = \bar{y} - \bar{x}a = 72.6 - 29.1 \times 3.7 = -36.4 \end{cases}$$

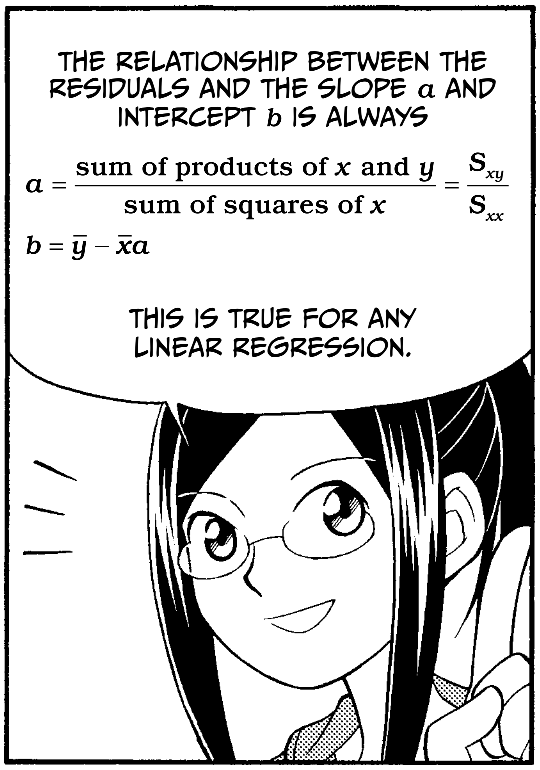
then the regression equation is

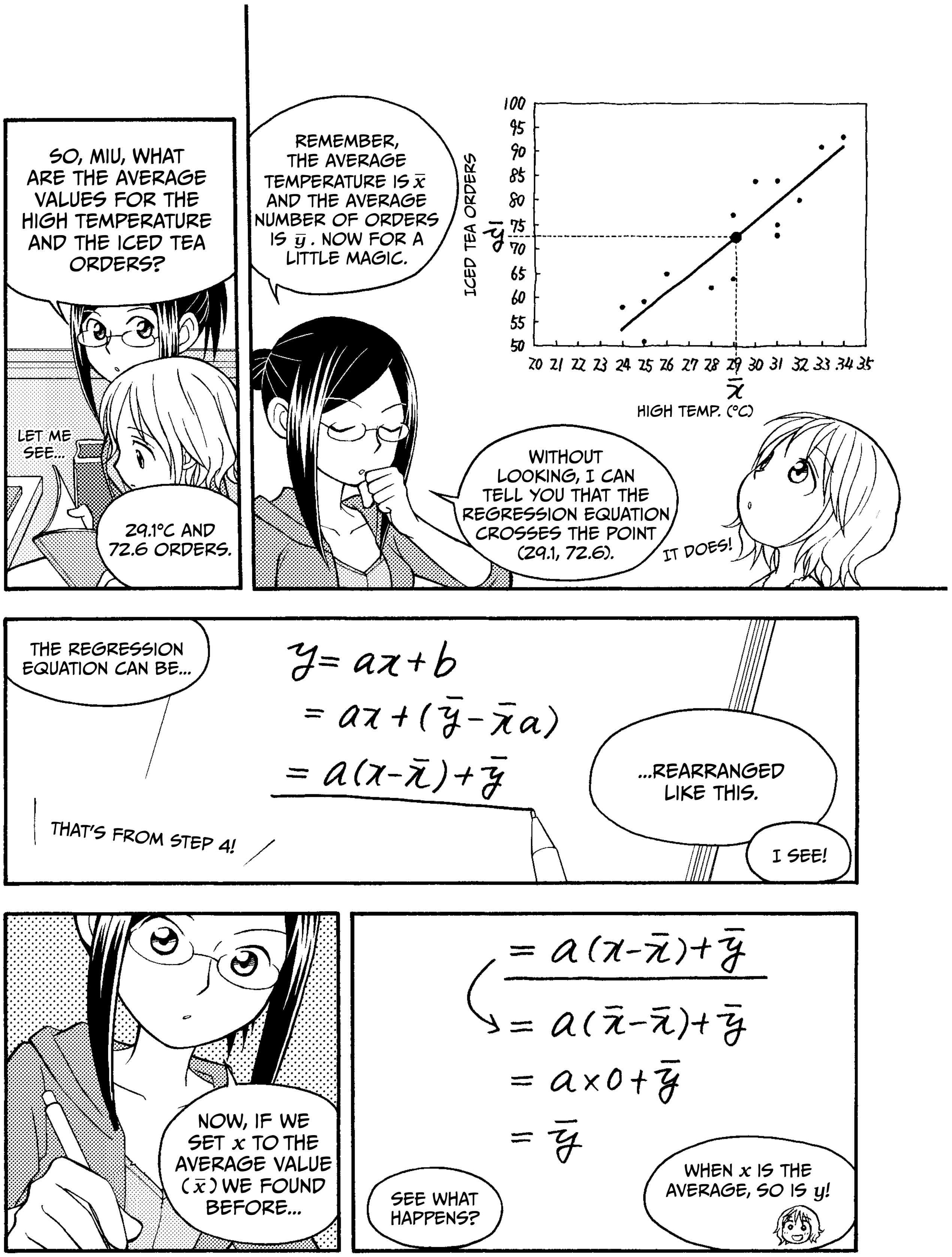
$$y = 3.7x - 36.4.$$

It's that simple!

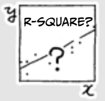
Note: The values shown are rounded for the sake of printing, but the result (36.4) was calculated using the full, unrounded values.

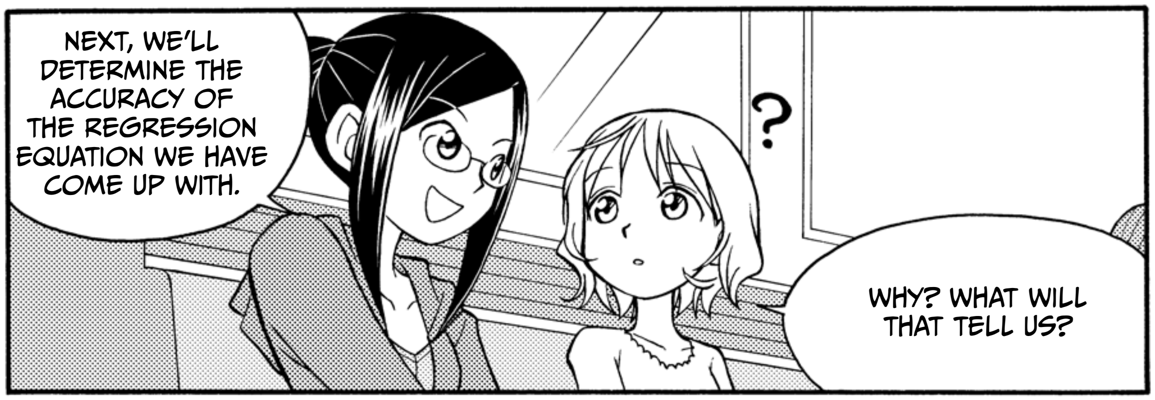


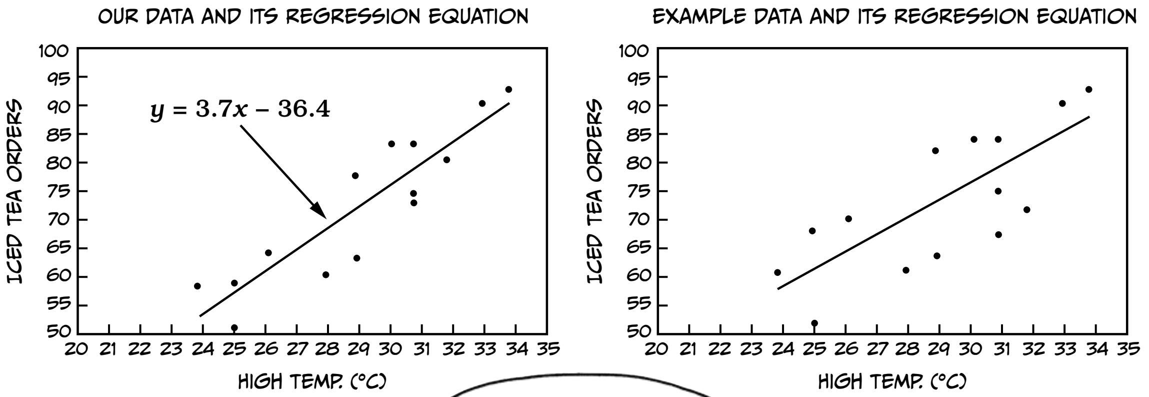

THE RELATIONSHIP BETWEEN THE RESIDUALS AND THE SLOPE a AND INTERCEPT b IS ALWAYS
$a = \frac{\text{sum of products of } x \text{ and } y}{\text{sum of squares of } x} = \frac{S_{xy}}{S_{xx}}$
$b = \bar{y} - \bar{x}a$
THIS IS TRUE FOR ANY LINEAR REGRESSION.

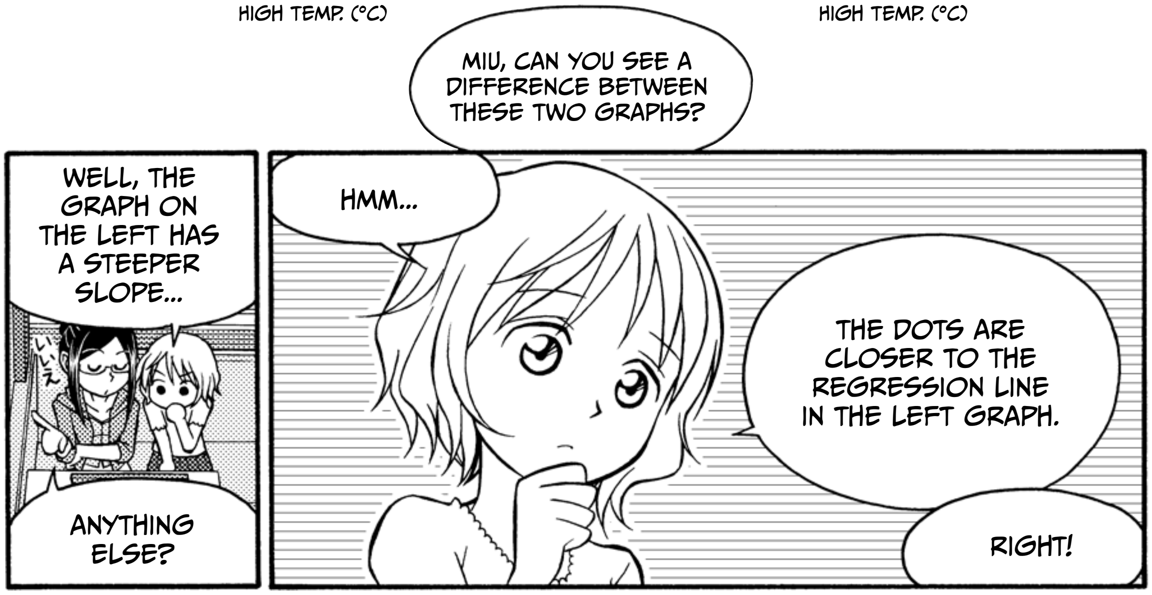

SO, MIU, WHAT ARE THE AVERAGE VALUES FOR THE HIGH TEMPERATURE AND THE ICED TEA ORDERS?
LET ME SEE...
29.1°C AND 72.6 ORDERS.
REMEMBER, THE AVERAGE TEMPERATURE IS $\bar{x}$ AND THE AVERAGE NUMBER OF ORDERS IS $\bar{y}$. NOW FOR A LITTLE MAGIC.
ICED TEA ORDERS
$\bar{y}$
HIGH TEMP. (°C)
$\bar{x}$
WITHOUT LOOKING, I CAN TELL YOU THAT THE REGRESSION EQUATION CROSSES THE POINT (29.1, 72.6).
IT DOES!
THE REGRESSION EQUATION CAN BE...
$y = ax + b$
$= ax + (\bar{y} - \bar{x}a)$
$= a(x - \bar{x}) + \bar{y}$
THAT'S FROM STEP 4!
...REARRANGED LIKE THIS.
I SEE!
NOW, IF WE SET x TO THE AVERAGE VALUE ($\bar{x}$) WE FOUND BEFORE...
$= a(x - \bar{x}) + \bar{y}$
$= a(\bar{x} - \bar{x}) + \bar{y}$
$= a \times 0 + \bar{y}$
$= \bar{y}$
SEE WHAT HAPPENS?
WHEN x IS THE AVERAGE, SO IS y!

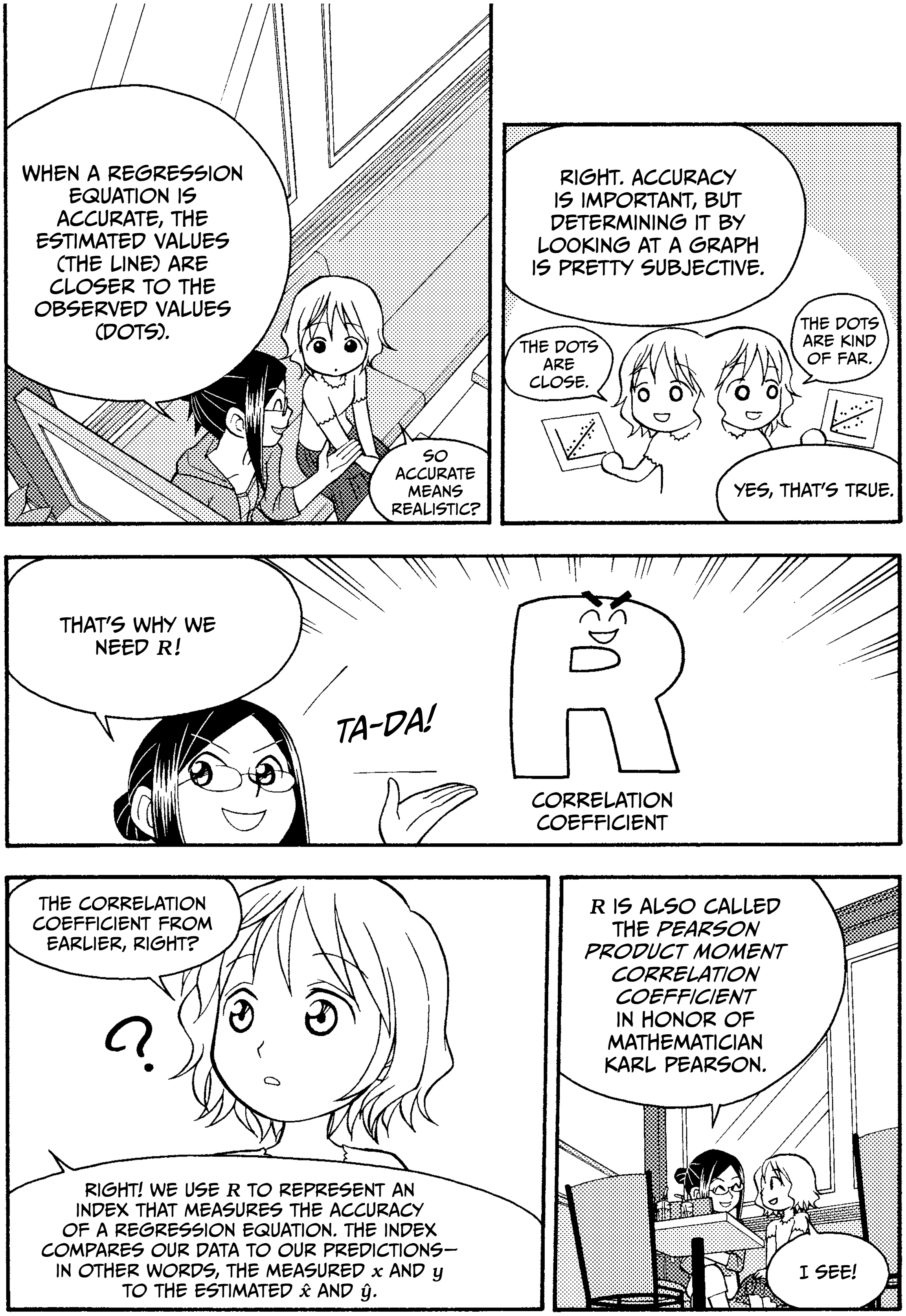
Step 3: Calculate the correlation coefficient (R) and assess our population and assumptions.

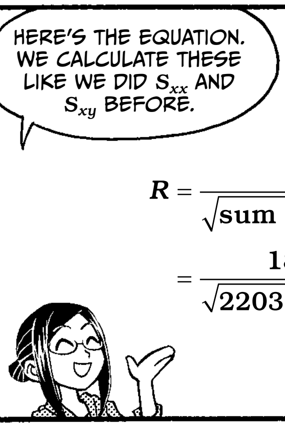



NEXT, WE'LL DETERMINE THE ACCURACY OF THE REGRESSION EQUATION WE HAVE COME UP WITH.
?
WHY? WHAT WILL THAT TELL US?



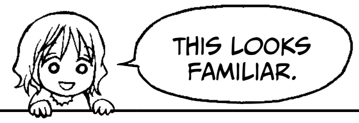

MIU, CAN YOU SEE A DIFFERENCE BETWEEN THESE TWO GRAPHS?
WELL, THE GRAPH ON THE LEFT HAS A STEEPER SLOPE...
いいえ
ANYTHING ELSE?
HMM...
THE DOTS ARE CLOSER TO THE REGRESSION LINE IN THE LEFT GRAPH.
RIGHT!


WHEN A REGRESSION EQUATION IS ACCURATE, THE ESTIMATED VALUES (THE LINE) ARE CLOSER TO THE OBSERVED VALUES (DOTS).
SO ACCURATE MEANS REALISTIC?
RIGHT. ACCURACY IS IMPORTANT, BUT DETERMINING IT BY LOOKING AT A GRAPH IS PRETTY SUBJECTIVE.
THE DOTS ARE KIND OF FAR.
THE DOTS ARE CLOSE.
YES, THAT'S TRUE.
THAT'S WHY WE NEED *R*!
TA-DA!
CORRELATION COEFFICIENT
R IS ALSO CALLED THE *PEARSON PRODUCT MOMENT CORRELATION COEFFICIENT* IN HONOR OF MATHEMATICIAN KARL PEARSON.
THE CORRELATION COEFFICIENT FROM EARLIER, RIGHT?
RIGHT! WE USE R TO REPRESENT AN INDEX THAT MEASURES THE ACCURACY OF A REGRESSION EQUATION. THE INDEX COMPARES OUR DATA TO OUR PREDICTIONS—IN OTHER WORDS, THE MEASURED x AND y TO THE ESTIMATED $\hat{x}$ AND $\hat{y}$.
I SEE!


HERE'S THE EQUATION. WE CALCULATE THESE LIKE WE DID S_{xx} AND S_{xy} BEFORE.

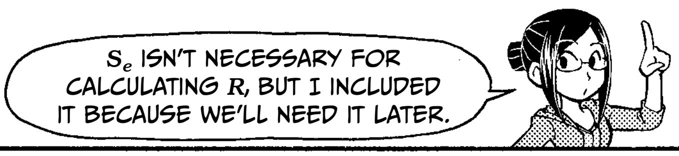
$$R = \frac{\text{sum of products } y \text{ and } \hat{y}}{\sqrt{\text{sum of squares of } y \times \text{sum of squares of } \hat{y}}} = \frac{S_{y\hat{y}}}{\sqrt{S_{yy} \times S_{\hat{y}\hat{y}}}}$$

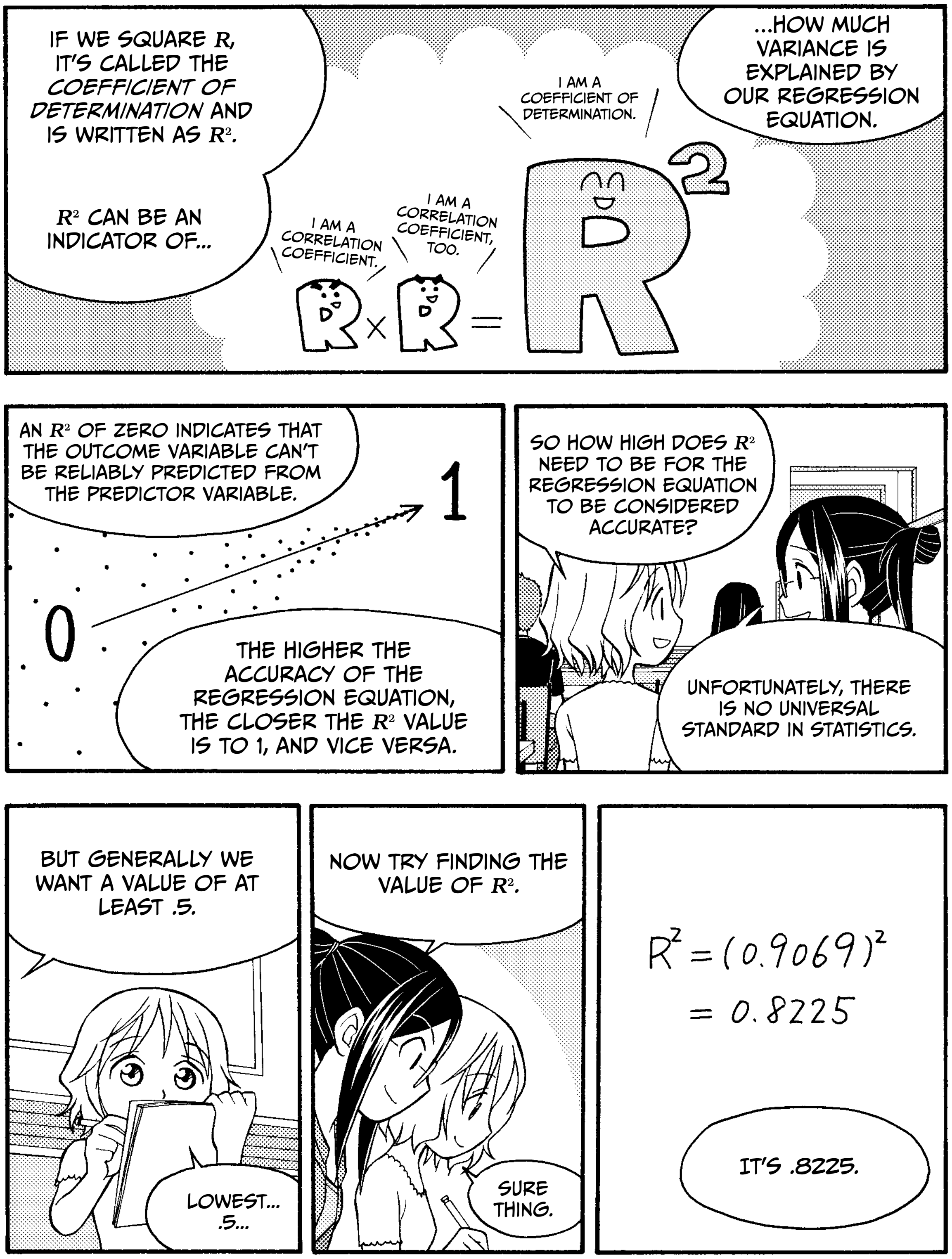
$$= \frac{1812.3}{\sqrt{2203.4 \times 1812.3}} = 0.9069$$

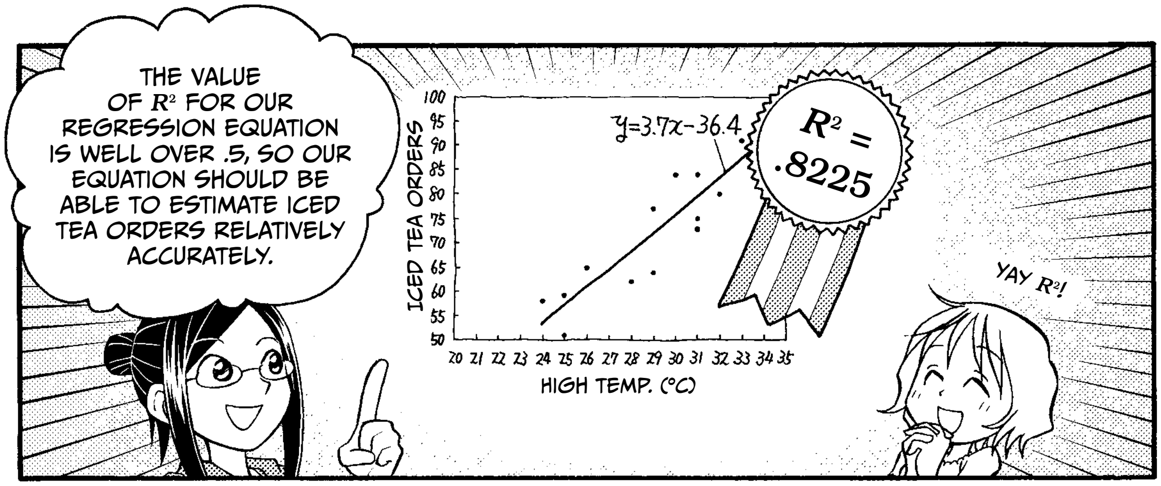


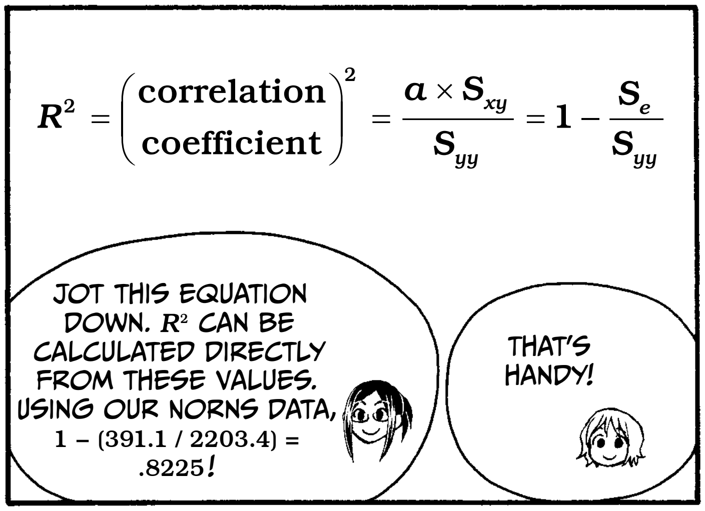
REGRESSION FUNCTION!

	Actual values y	Estimated values $\hat{y} = 3.7x - 36.4$	$y - \bar{y}$	$\hat{y} - \bar{\hat{y}}$	$(y - \bar{y})^2$	$(\hat{y} - \bar{\hat{y}})^2$	$(y - \bar{y})(\hat{y} - \bar{\hat{y}})$	$(y - \hat{y})^2$
22nd (Mon.)	77	72.0	4.4	−0.5	19.6	0.3	−2.4	24.6
23rd (Tues.)	62	68.3	−10.6	−4.3	111.8	18.2	45.2	39.7
24th (Wed.)	93	90.7	20.4	18.2	417.3	329.6	370.9	5.2
25th (Thurs.)	84	79.5	11.4	6.9	130.6	48.2	79.3	20.1
26th (Fri.)	59	57.1	−13.6	−15.5	184.2	239.8	210.2	3.7
27th (Sat.)	64	72.0	−8.6	−0.5	73.5	0.3	4.6	64.6
28th (Sun.)	80	83.3	7.4	10.7	55.2	114.1	79.3	10.6
29th (Mon.)	75	79.5	2.4	6.9	5.9	48.2	16.9	20.4
30th (Tues.)	58	53.3	−14.6	−19.2	212.3	369.5	280.1	21.6
31st (Wed.)	91	87.0	18.4	14.4	339.6	207.9	265.7	16.1
1st (Thurs.)	51	57.1	−21.6	−15.5	465.3	239.8	334.0	37.0
2nd (Fri.)	73	79.5	0.4	6.9	0.2	48.2	3.0	42.4
3rd (Sat.)	65	60.8	−7.6	−11.7	57.3	138.0	88.9	17.4
4th (Sun.)	84	75.8	11.4	3.2	130.6	10.3	36.6	67.6
Sum	1016	1016	0	0	2203.4	1812.3	1812.3	391.1
Average	72.6	72.6						
	$\downarrow$ $\bar{y}$	$\downarrow$ $\bar{\hat{y}}$			$\downarrow$ S_{yy}	$\downarrow$ $S_{\hat{y}\hat{y}}$	$\downarrow$ $S_{y\hat{y}}$	$\downarrow$ S_e


S_e ISN'T NECESSARY FOR CALCULATING R, BUT I INCLUDED IT BECAUSE WE'LL NEED IT LATER.

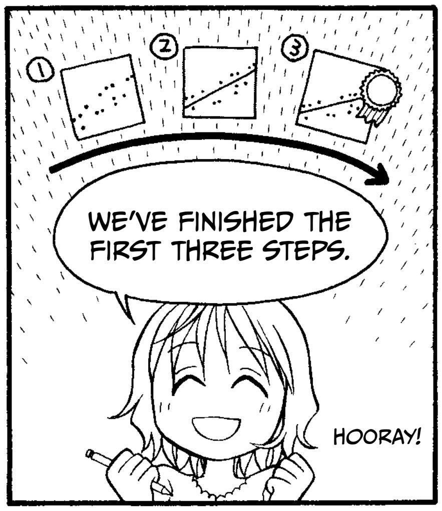

IF WE SQUARE R, IT'S CALLED THE *COEFFICIENT OF DETERMINATION* AND IS WRITTEN AS R^2.
R^2 CAN BE AN INDICATOR OF...
I AM A CORRELATION COEFFICIENT.
I AM A CORRELATION COEFFICIENT, TOO.
I AM A COEFFICIENT OF DETERMINATION.
$R \times R = R^2$
...HOW MUCH VARIANCE IS EXPLAINED BY OUR REGRESSION EQUATION.
AN R^2 OF ZERO INDICATES THAT THE OUTCOME VARIABLE CAN'T BE RELIABLY PREDICTED FROM THE PREDICTOR VARIABLE.
0
1
THE HIGHER THE ACCURACY OF THE REGRESSION EQUATION, THE CLOSER THE R^2 VALUE IS TO 1, AND VICE VERSA.
SO HOW HIGH DOES R^2 NEED TO BE FOR THE REGRESSION EQUATION TO BE CONSIDERED ACCURATE?
UNFORTUNATELY, THERE IS NO UNIVERSAL STANDARD IN STATISTICS.
BUT GENERALLY WE WANT A VALUE OF AT LEAST .5.
LOWEST... .5...
NOW TRY FINDING THE VALUE OF R^2.
SURE THING.
$R^2 = (0.9069)^2 = 0.8225$
IT'S .8225.


THE VALUE OF R^2 FOR OUR REGRESSION EQUATION IS WELL OVER .5, SO OUR EQUATION SHOULD BE ABLE TO ESTIMATE ICED TEA ORDERS RELATIVELY ACCURATELY.
ICED TEA ORDERS
HIGH TEMP. (°C)
y=3.7x−36.4
R^2 = .8225
YAY R^2!

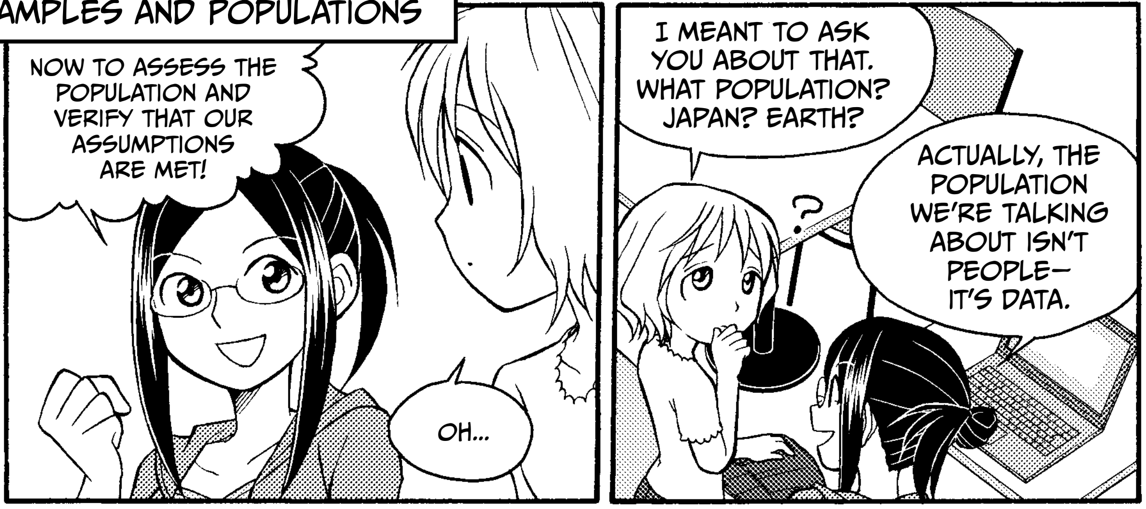


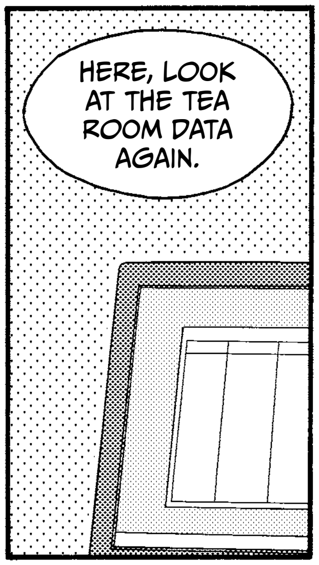
$$R^2 = \left(\begin{matrix}\textbf{correlation}\\ \textbf{coefficient}\end{matrix}\right)^2 = \frac{a \times S_{xy}}{S_{yy}} = 1 - \frac{S_e}{S_{yy}}$$

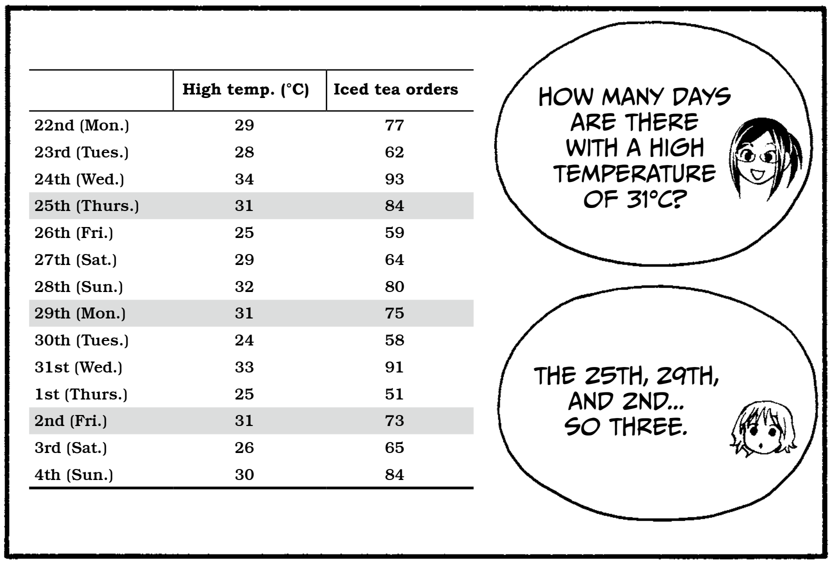
JOT THIS EQUATION DOWN. R^2 CAN BE CALCULATED DIRECTLY FROM THESE VALUES. USING OUR NORNS DATA, 1 − (391.1 / 2203.4) = .8225!
THAT'S HANDY!



SAMPLES AND POPULATIONS

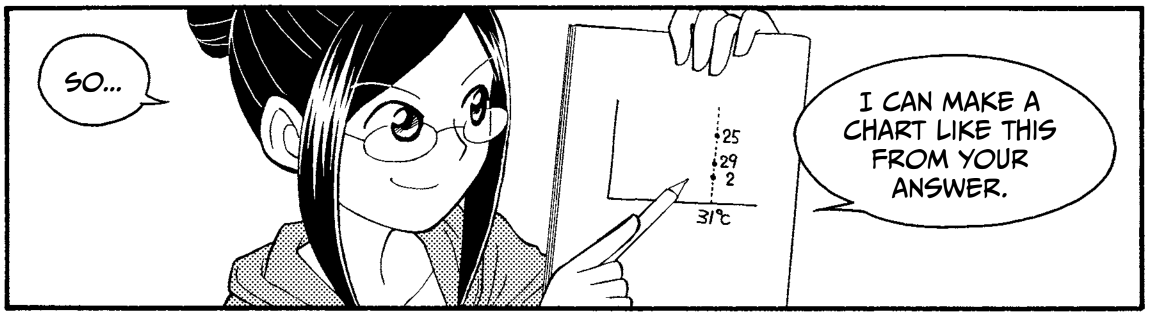

I MEANT TO ASK YOU ABOUT THAT. WHAT POPULATION? JAPAN? EARTH?
ACTUALLY, THE POPULATION WE'RE TALKING ABOUT ISN'T PEOPLE— IT'S DATA.
NOW TO ASSESS THE POPULATION AND VERIFY THAT OUR ASSUMPTIONS ARE MET!
OH...

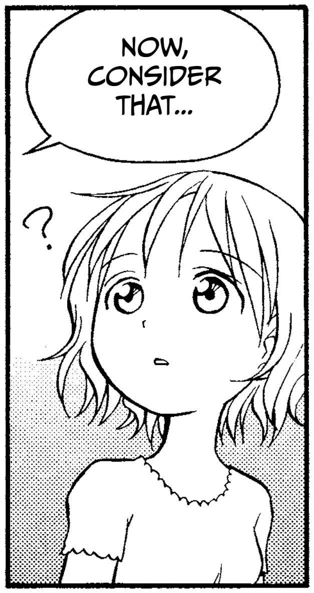


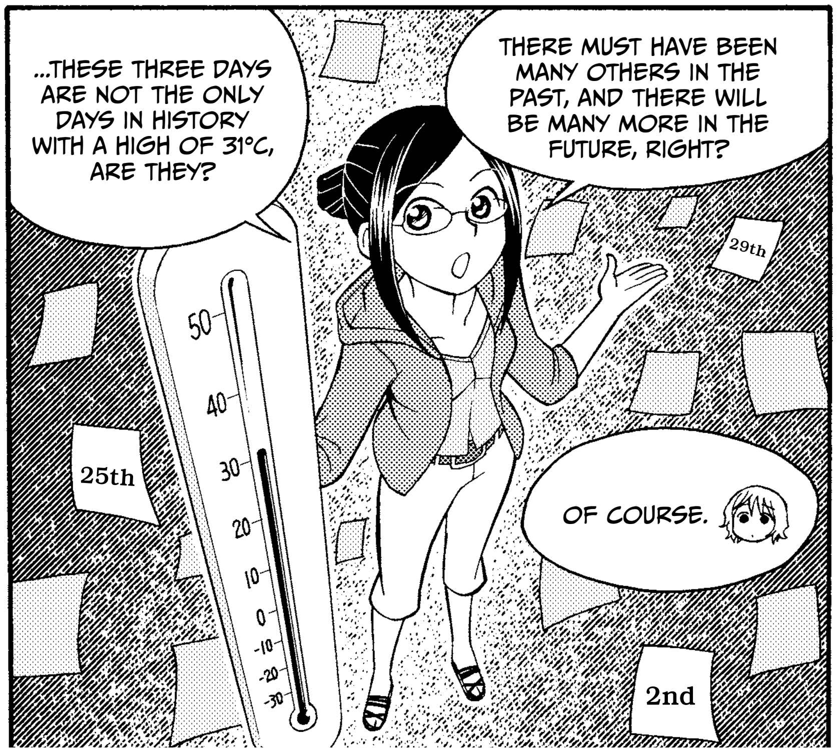


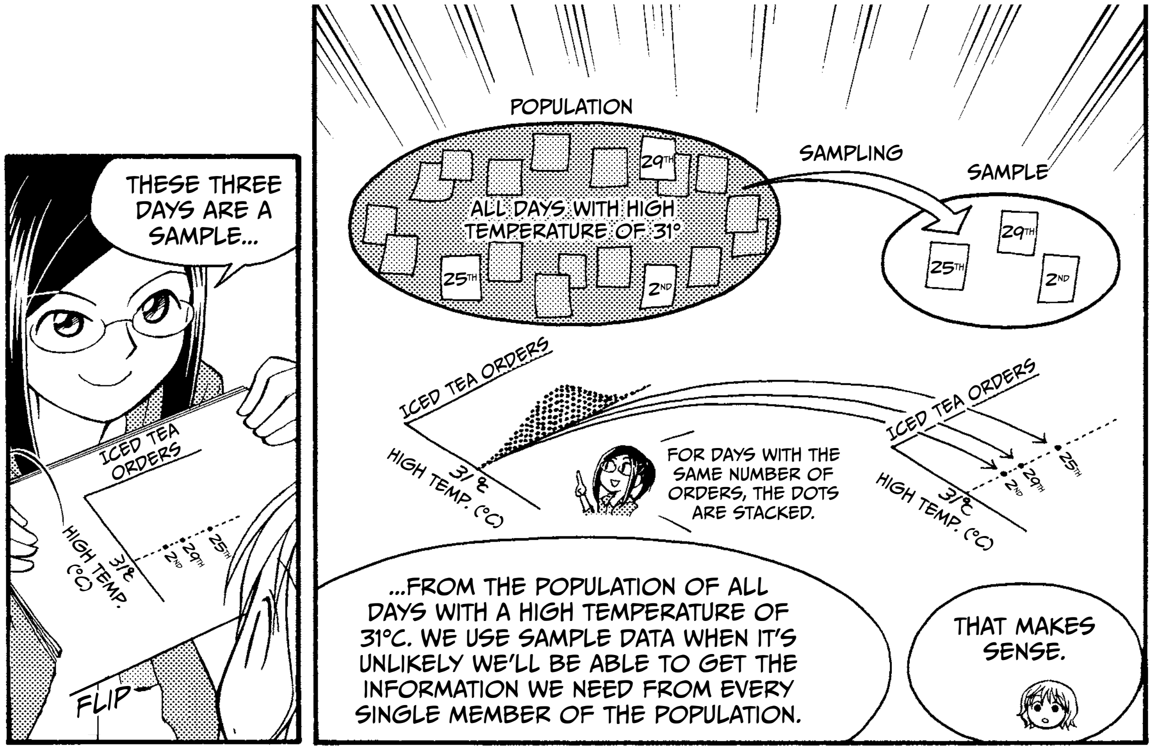
	High temp. (°C)	Iced tea orders
22nd (Mon.)	29	77
23rd (Tues.)	28	62
24th (Wed.)	34	93
25th (Thurs.)	31	84
26th (Fri.)	25	59
27th (Sat.)	29	64
28th (Sun.)	32	80
29th (Mon.)	31	75
30th (Tues.)	24	58
31st (Wed.)	33	91
1st (Thurs.)	25	51
2nd (Fri.)	31	73
3rd (Sat.)	26	65
4th (Sun.)	30	84

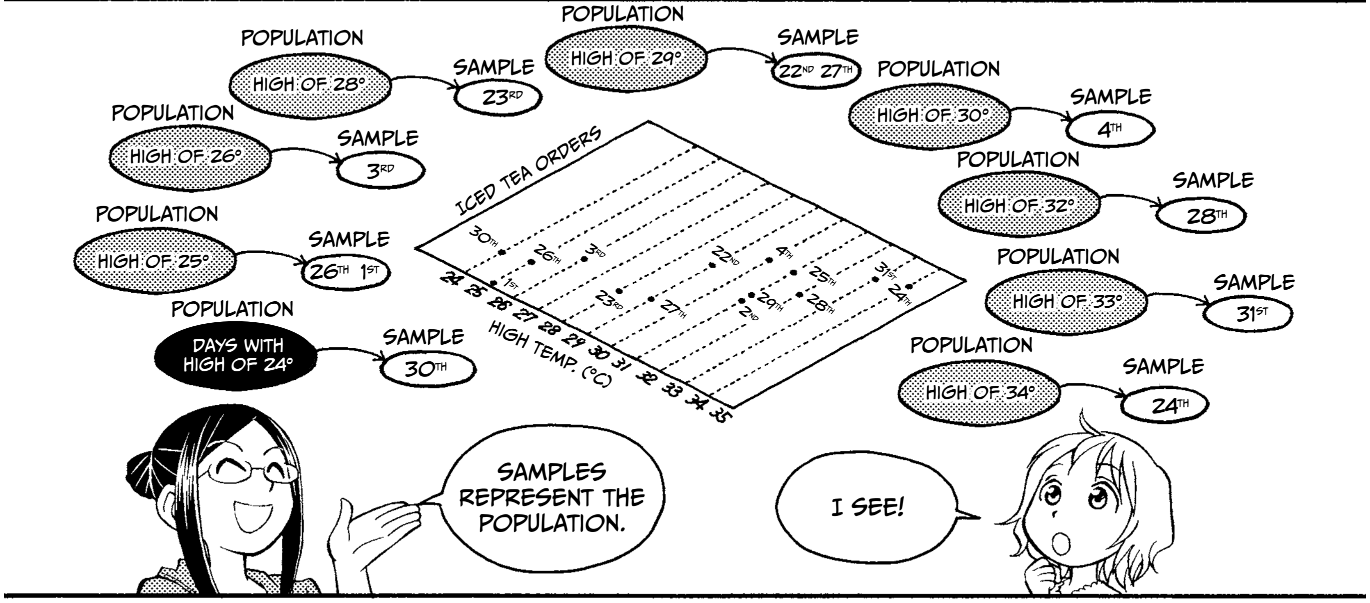
HOW MANY DAYS ARE THERE WITH A HIGH TEMPERATURE OF 31°C?
THE 25TH, 29TH, AND 2ND... SO THREE.

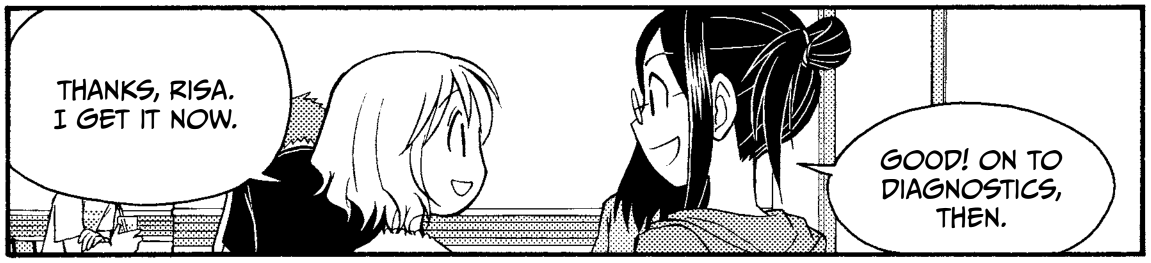





...THESE THREE DAYS ARE NOT THE ONLY DAYS IN HISTORY WITH A HIGH OF 31°C, ARE THEY?
THERE MUST HAVE BEEN MANY OTHERS IN THE PAST, AND THERE WILL BE MANY MORE IN THE FUTURE, RIGHT?
29th
25th
OF COURSE.
2nd


POPULATION
ALL DAYS WITH HIGH TEMPERATURE OF 31°
29th
25th
2nd
SAMPLING
SAMPLE
THESE THREE DAYS ARE A SAMPLE...
ICED TEA ORDERS
HIGH TEMP. (°C)
31°C
FOR DAYS WITH THE SAME NUMBER OF ORDERS, THE DOTS ARE STACKED.
FLIP
...FROM THE POPULATION OF ALL DAYS WITH A HIGH TEMPERATURE OF 31°C. WE USE SAMPLE DATA WHEN IT'S UNLIKELY WE'LL BE ABLE TO GET THE INFORMATION WE NEED FROM EVERY SINGLE MEMBER OF THE POPULATION.
THAT MAKES SENSE.

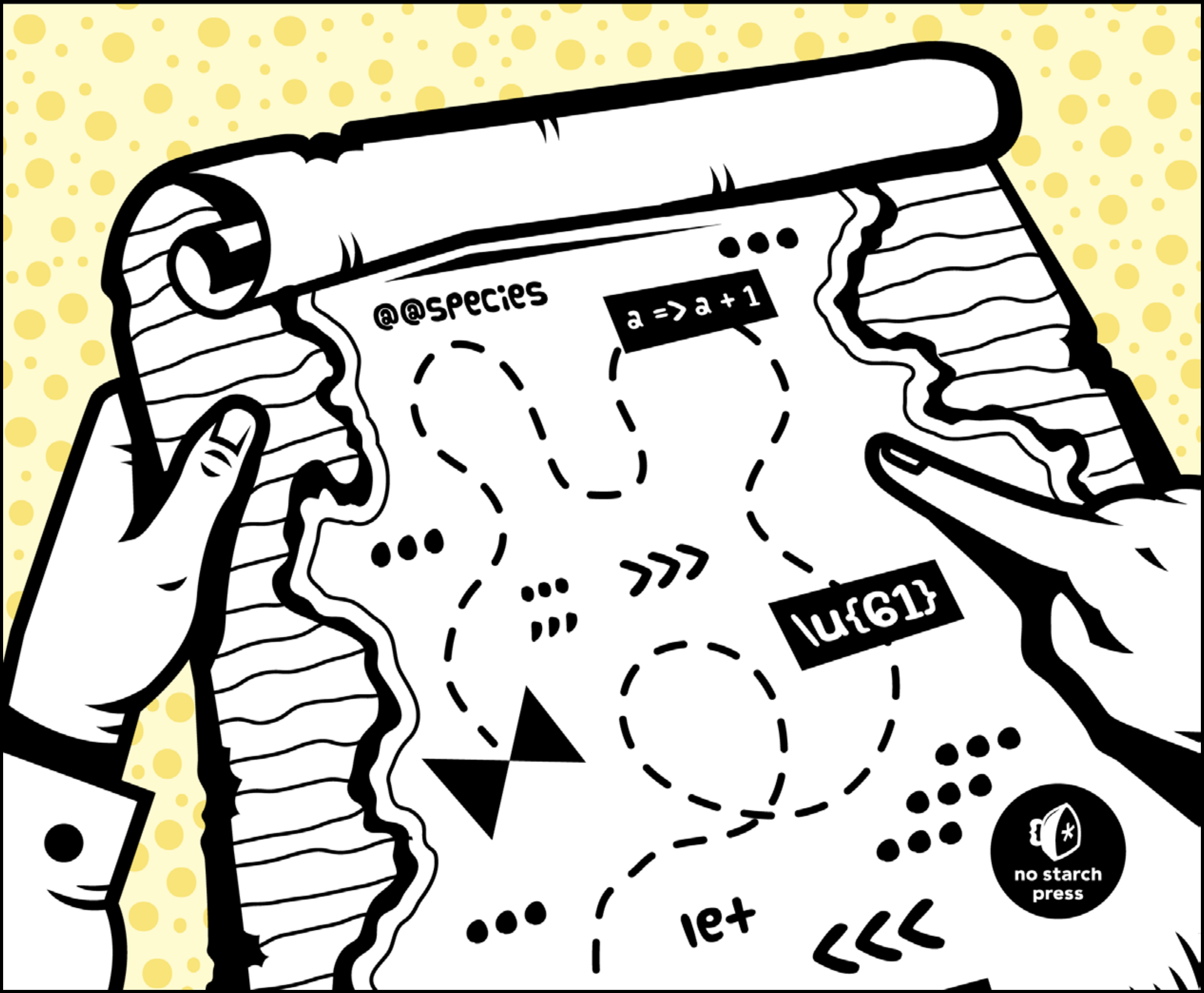

POPULATION
HIGH OF 29°
SAMPLE
22nd 27th
POPULATION
HIGH OF 28°
SAMPLE
23rd
POPULATION
HIGH OF 30°
SAMPLE
4th
POPULATION
HIGH OF 26°
SAMPLE
3rd
POPULATION
HIGH OF 32°
SAMPLE
28th
POPULATION
HIGH OF 25°
SAMPLE
26th 1st
POPULATION
HIGH OF 33°
SAMPLE
31st
POPULATION
DAYS WITH HIGH OF 24°
SAMPLE
30th
POPULATION
HIGH OF 34°
SAMPLE
24th
ICED TEA ORDERS
HIGH TEMP. (°C)
24 25 26 27 28 29 30 31 32 33 34 35
30th 26th 3rd 1st 23rd 27th 22nd 4th 2nd 29th 25th 28th 31st 24th
I SEE!
SAMPLES REPRESENT THE POPULATION.


GOOD! ON TO DIAGNOSTICS, THEN.
THANKS, RISA. I GET IT NOW.

UNDERSTANDING ECMASCRIPT 6

THE *DEFINITIVE GUIDE* FOR JAVASCRIPT DEVELOPERS

NICHOLAS C. ZAKAS



4
EXPANDED OBJECT FUNCTIONALITY

ECMAScript 6 focuses heavily on making objects more useful, which makes sense because nearly every value in JavaScript is some type of object. The number of objects developers use in an average JavaScript program continues to increase as the complexity of JavaScript applications increases. With more objects in a program, it has become necessary to use them more effectively.

ECMAScript 6 improves the use of objects in a number of ways, from simple syntax extensions to options for manipulating and interacting with them, and this chapter covers those improvements in detail.

Object Categories

JavaScript uses different terminology to describe objects in the standard as opposed to those added by execution environments, such as the browser. The ECMAScript 6 specification has clear definitions for each object category. It's essential to understand this terminology to grasp the language as a whole. The object categories are:

Ordinary objects Have all the default internal behaviors for objects in JavaScript.

Exotic objects Have internal behavior that differs from the default in some way.

Standard objects Defined by ECMAScript 6, such as `Array`, `Date`, and so on. Standard objects can be ordinary or exotic.

Built-in objects Present in a JavaScript execution environment when a script begins to execute. All standard objects are built-in objects.

I'll use these terms throughout the book to explain the various objects that ECMAScript 6 defines.

Object Literal Syntax Extensions

The object literal is one of the most popular patterns in JavaScript. JSON is built on its syntax, and it's in nearly every JavaScript file on the Internet. The object literal's popularity is due to its succinct syntax for creating objects that would otherwise take several lines of code to create. Fortunately for developers, ECMAScript 6 makes object literals more powerful and even more succinct by extending the syntax in several ways.

Property Initializer Shorthand

In ECMAScript 5 and earlier, object literals were simply collections of name-value pairs, meaning that some duplication could occur when property values are initialized. For example:

```
function createPerson(name, age) {
    return {
        name: name,
        age: age
    };
}
```

The `createPerson()` function creates an object whose property names are the same as the function parameter names. The result appears to be the duplication of `name` and `age`, even though one is the name of an object property and the other provides the value of that property. The key `name` in the returned object is assigned the value contained in the variable `name`, and the key `age` in the returned object is assigned the value contained in the variable `age`.

In ECMAScript 6, you can eliminate the duplication that exists around property names and local variables by using the *property initializer* shorthand syntax. When an object property name is the same as the local variable name, you can simply include the name without a colon and value. For example, `createPerson()` can be rewritten for ECMAScript 6 as follows:

```
function createPerson(name, age) {
    return {
        name,
        age
    };
}
```

When a property in an object literal only has a name, the JavaScript engine looks in the surrounding scope for a variable of the same name. If it finds one, that variable's value is assigned to the same name on the object literal. In this example, the object literal property `name` is assigned the value of the local variable `name`.

Shorthand property syntax makes object literal initialization even more succinct and helps to eliminate naming errors. Assigning a property with the same name as a local variable is a very common pattern in JavaScript, making this extension a welcome addition.

Concise Methods

ECMAScript 6 also improves the syntax for assigning methods to object literals. In ECMAScript 5 and earlier, you must specify a name and then the full function definition to add a method to an object, as follows:

```
var person = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};
```

In ECMAScript 6, the syntax is made more concise by eliminating the colon and the `function` keyword. That means you can rewrite the example like this:

```
var person = {
    name: "Nicholas",
    sayName() {
        console.log(this.name);
    }
};
```

This shorthand syntax, also called *concise method* syntax, creates a method on the `person` object just as the previous example did. The `sayName()` property is assigned an anonymous function expression and has all the same characteristics as the ECMAScript 5 `sayName()` function. The one difference is that concise methods can use `super` (discussed in "Easy Prototype Access with Super References" on page 139), whereas the nonconcise methods cannot.

NOTE *The* `name` *property of a method created using concise method shorthand is the name used before the parentheses. In this example, the* `name` *property for* `person.sayName()` *is* `"sayName"`.

Computed Property Names

ECMAScript 5 and earlier could compute property names on object instances when those properties were set with square brackets instead of dot notation. The square brackets allow you to specify property names using variables and string literals that might contain characters that would cause a syntax error if they were used in an identifier. Here's an example:

```
var person = {},
    lastName = "last name";

person["first name"] = "Nicholas";
person[lastName] = "Zakas";

console.log(person["first name"]);      // "Nicholas"
console.log(person[lastName]);          // "Zakas"
```

Because `lastName` is assigned a value of `"last name"`, both property names in this example use a space, making it impossible to reference them using dot notation. However, bracket notation allows any string value to be used as a property name, so assigning `"first name"` to `"Nicholas"` and `"last name"` to `"Zakas"` works.

Additionally, you can use string literals directly as property names in object literals, like this:

```
var person = {
    "first name": "Nicholas"
};

console.log(person["first name"]);      // "Nicholas"
```

This pattern works for property names that are known ahead of time and can be represented with a string literal. However, if the property name `"first name"` were contained in a variable (as in the previous example) or had to be calculated, there would be no way to define that property using an object literal in ECMAScript 5.

In ECMAScript 6, computed property names are part of the object literal syntax, and they use the same square bracket notation that has been used to reference computed property names in object instances. For example:

```
let lastName = "last name";

let person = {
    "first name": "Nicholas",
    [lastName]: "Zakas"
};

console.log(person["first name"]);      // "Nicholas"
console.log(person[lastName]);          // "Zakas"
```

The square brackets inside the object literal indicate that the property name is computed, so its contents are evaluated as a string. That means you can also include expressions, such as the following:

```
var suffix = " name";

var person = {
    ["first" + suffix]: "Nicholas",
    ["last" + suffix]: "Zakas"
};

console.log(person["first name"]);      // "Nicholas"
console.log(person["last name"]);       // "Zakas"
```

These properties evaluate to `"first name"` and `"last name"`, and you can use those strings to reference the properties later. Anything you would put inside square brackets while using bracket notation on object instances will also work for computed property names inside object literals.

New Methods

One of the design goals of ECMAScript, beginning with ECMAScript 5, was to avoid both creating new global functions and creating methods on `Object.prototype`. Instead, when the developers want to add new methods to the standard, they make those methods available on an appropriate existing object. As a result, the `Object` global has received an increasing number of methods when no other objects are more appropriate. ECMAScript 6 introduces a couple of new methods on the `Object` global that are designed to make certain tasks easier.

The Object.is() Method

When you want to compare two values in JavaScript, you're probably used to using either the equals operator (`==`) or the identically equals operator (`===`). Many developers prefer the latter to avoid type coercion during comparison. But even the identically equals operator isn't entirely accurate. For example, the values +0 and −0 are considered equal by `===`, even though they're represented differently in the JavaScript engine. Also, `NaN === NaN` returns `false`, which necessitates using `isNaN()` to detect `NaN` properly.

ECMAScript 6 introduces the `Object.is()` method to remedy the remaining inaccuracies of the identically equals operator. This method accepts two arguments and returns `true` if the values are equivalent. Two values are considered equivalent when they're the same type and have the same value. Here are some examples:

```
console.log(+0 == -0);              // true
console.log(+0 === -0);             // true
console.log(Object.is(+0, -0));     // false

console.log(NaN == NaN);            // false
console.log(NaN === NaN);           // false
console.log(Object.is(NaN, NaN));   // true

console.log(5 == 5);                // true
console.log(5 == "5");              // true
console.log(5 === 5);               // true
console.log(5 === "5");             // false
console.log(Object.is(5, 5));       // true
console.log(Object.is(5, "5"));     // false
```

In many cases, `Object.is()` works the same as the `===` operator. The only differences are that +0 and −0 are considered not equivalent, and `NaN` is considered equivalent to `NaN`. But there's no need to stop using equality operators. Choose whether to use `Object.is()` instead of `==` or `===` based on how those special cases affect your code.

The Object.assign() Method

Mixins are among the most popular patterns for object composition in JavaScript. In a mixin, one object receives properties and methods from another object. Many JavaScript libraries have a mixin method similar to this:

```
function mixin(receiver, supplier) {
    Object.keys(supplier).forEach(function(key) {
        receiver[key] = supplier[key];
    });

    return receiver;
}
```

The `mixin()` function iterates over the own properties of `supplier` and copies them onto `receiver` (a shallow copy, where object references are shared when property values are objects). This allows the `receiver` to gain new properties without inheritance, as in this code:

```
function EventTarget() { /*...*/ }
EventTarget.prototype = {
    constructor: EventTarget,
    emit: function() { /*...*/ },
    on: function() { /*...*/ }
};

var myObject = {};
mixin(myObject, EventTarget.prototype);

myObject.emit("somethingChanged");
```

Here, `myObject` receives behavior from the `EventTarget.prototype` object. This gives `myObject` the ability to publish events and subscribe to them using the `emit()` and `on()` methods, respectively.

This mixin pattern became popular enough that ECMAScript 6 added the `Object.assign()` method, which behaves the same way, accepting a receiver and any number of suppliers and then returning the receiver. The name change from `mixin()` to `assign()` reflects the actual operation that occurs. Because the `mixin()` function uses the assignment operator (`=`), it cannot copy accessor properties to the receiver as accessor properties. The name `Object.assign()` was chosen to reflect this distinction.

NOTE *Similar methods in various libraries might have other names for the same basic functionality; popular alternates include the `extend()` and `mix()` methods. In addition to the `Object.assign()` method, an `Object.mixin()` method was briefly added in ECMAScript 6. The primary difference was that `Object.mixin()` also copied over accessor properties, but the method was removed due to concerns over the use of `super` (discussed in "Easy Prototype Access with Super References" on page 139).*

You can use `Object.assign()` anywhere you would have used the `mixin()` function. Here's an example:

```
function EventTarget() { /*...*/ }
EventTarget.prototype = {
    constructor: EventTarget,
    emit: function() { /*...*/ },
    on: function() { /*...*/ }
}

var myObject = {}
Object.assign(myObject, EventTarget.prototype);

myObject.emit("somethingChanged");
```

The `Object.assign()` method accepts any number of suppliers, and the receiver receives the properties in the order in which the suppliers are specified. That means the second supplier might overwrite a value from the first supplier on the receiver, which is what happens in this code snippet:

```
var receiver = {};

Object.assign(receiver,
    {
        type: "js",
        name: "file.js"
    },
    {
        type: "css"
    }
);

console.log(receiver.type);     // "css"
console.log(receiver.name);     // "file.js"
```

The value of `receiver.type` is `"css"` because the second supplier overwrote the value of the first.

The `Object.assign()` method isn't a significant addition to ECMAScript 6, but it does formalize a common function found in many JavaScript libraries.

WORKING WITH ACCESSOR PROPERTIES

Keep in mind that `Object.assign()` doesn't create accessor properties on the receiver when a supplier has accessor properties. Because `Object.assign()` uses the assignment operator, an accessor property on a supplier will become a data property on the receiver. For example:

```
var receiver = {},
    supplier = {
        get name() {
            return "file.js"
        }
    };

Object.assign(receiver, supplier);

var descriptor = Object.getOwnPropertyDescriptor(receiver, "name");

console.log(descriptor.value);      // "file.js"
console.log(descriptor.get);        // undefined
```

In this code, the `supplier` has an accessor property called `name`. After using the `Object.assign()` method, `receiver.name` exists as a data property with a value of `"file.js"` because `supplier.name` returned `"file.js"` when `Object.assign()` was called.

Duplicate Object Literal Properties

ECMAScript 5 strict mode introduced a check for duplicate object literal properties that would throw an error if a duplicate was found. For example, this code was problematic:

```
"use strict";

var person = {
    name: "Nicholas",
    name: "Greg"       // syntax error in ES5 strict mode
};
```

When running in ECMAScript 5 strict mode, the second `name` property causes a syntax error. But in ECMAScript 6, the duplicate property check was removed. Strict and non-strict mode code no longer check for duplicate properties. Instead, the last property of the given name becomes the property's actual value, as shown here:

```
"use strict";

var person = {
    name: "Nicholas",
    name: "Greg"       // no error in ES6 strict mode
};

console.log(person.name);       // "Greg"
```

In this example, the value of `person.name` is `"Greg"` because that's the last value assigned to the property.

Own Property Enumeration Order

ECMAScript 5 didn't define the enumeration order of object properties; the JavaScript engine vendors did. However, ECMAScript 6 strictly defines the order in which own properties must be returned when they're enumerated. This affects how properties are returned using `Object.getOwnPropertyNames()` and `Reflect.ownKeys` (covered in Chapter 12). It also affects the order in which properties are processed by `Object.assign()`.

The basic order for own property enumeration is:

1. All numeric keys in ascending order
2. All string keys in the order in which they were added to the object
3. All symbol keys (covered in Chapter 6) in the order in which they were added to the object

Here's an example:

```
var obj = {
    a: 1,
    0: 1,
    c: 1,
    2: 1,
    b: 1,
    1: 1
};

obj.d = 1;

console.log(Object.getOwnPropertyNames(obj).join(""));     // "012acbd"
```

The `Object.getOwnPropertyNames()` method returns the properties in `obj` in the order `0`, `1`, `2`, `a`, `c`, `b`, `d`. Note that the numeric keys are grouped together and sorted, even though they appear out of order in the object literal. The string keys come after the numeric keys and appear in the order in which they were added to `obj`. The keys in the object literal come first, followed by any dynamic keys that were added later (in this case, `d`).

NOTE *The `for-in` loop still has an unspecified enumeration order because not all JavaScript engines implement it the same way. The `Object.keys()` method and `JSON.stringify()` are both specified to use the same (unspecified) enumeration order as `for-in`.*

Although enumeration order is a subtle change to how JavaScript works, it's not uncommon to find programs that rely on a specific enumeration order to work correctly. ECMAScript 6, by defining the enumeration order, ensures that JavaScript code relying on enumeration will work correctly regardless of where it is executed.

Enhancements for Prototypes

Prototypes are the foundation of inheritance in JavaScript, and ECMAScript 6 continues to make prototypes more useful. Early versions of JavaScript severely limited what you could do with prototypes. However, as the language matured and developers became more familiar with how prototypes work, it became clear that developers wanted more control over prototypes and easier ways to work with them. As a result, ECMAScript 6 introduced some improvements to prototypes.

Changing an Object's Prototype

Normally, an object's prototype is specified when the object is created, via either a constructor or the `Object.create()` method. The idea that an object's prototype remains unchanged after instantiation was one of the predominant assumptions in JavaScript programming through ECMAScript 5. ECMAScript 5 did add the `Object.getPrototypeOf()` method for retrieving the prototype of any given object, but it still lacked a standard way to change an object's prototype after instantiation.

ECMAScript 6 changes that assumption with the addition of the `Object.setPrototypeOf()` method, which allows you to change the prototype of any given object. The `Object.setPrototypeOf()` method accepts two arguments: the object whose prototype should change and the object that should become the first argument's prototype. For example:

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

let dog = {
    getGreeting() {
        return "Woof";
    }
};

// prototype is person
let friend = Object.create(person);
console.log(friend.getGreeting());                      // "Hello"
console.log(Object.getPrototypeOf(friend) === person);  // true

// set prototype to dog
Object.setPrototypeOf(friend, dog);
console.log(friend.getGreeting());                      // "Woof"
console.log(Object.getPrototypeOf(friend) === dog);     // true
```

This code defines two base objects: `person` and `dog`. Both objects have a `getGreeting()` method that returns a string. The object `friend` first inherits from the `person` object, meaning that `getGreeting()` outputs `"Hello"`. When the prototype becomes the `dog` object, `person.getGreeting()` outputs `"Woof"` because the original relationship to `person` is broken.

The actual value of an object's prototype is stored in an internal-only property called `[[Prototype]]`. The `Object.getPrototypeOf()` method returns the value stored in `[[Prototype]]` and `Object.setPrototypeOf()` changes the value stored in `[[Prototype]]`. However, these aren't the only ways to work with the `[[Prototype]]` value.

Easy Prototype Access with Super References

As previously mentioned, prototypes are very important in JavaScript, and a lot of work went into making them easier to use in ECMAScript 6. Another improvement is the introduction of `super` references, which make accessing functionality on an object's prototype easier. For example, to override a method on an object instance so it also calls the prototype method of the same name, you'd do the following in ECMAScript 5:

```
let person = {
    getGreeting() {
        return "Hello";
    }
};
```

```
let dog = {
    getGreeting() {
        return "Woof";
    }
};


let friend = {
    getGreeting() {
        return Object.getPrototypeOf(this).getGreeting.call(this) + ", hi!";
    }
};

// set prototype to person
Object.setPrototypeOf(friend, person);
console.log(friend.getGreeting());                      // "Hello, hi!"
console.log(Object.getPrototypeOf(friend) === person);  // true

// set prototype to dog
Object.setPrototypeOf(friend, dog);
console.log(friend.getGreeting());                      // "Woof, hi!"
console.log(Object.getPrototypeOf(friend) === dog);     // true
```

In this example, `getGreeting()` on `friend` calls the prototype method of the same name. The `Object.getPrototypeOf()` method ensures the correct prototype is called, and then an additional string is appended to the output. The additional `.call(this)` ensures that the `this` value inside the prototype method is set correctly.

Remembering to use `Object.getPrototypeOf()` and `.call(this)` to call a method on the prototype is a bit involved, so ECMAScript 6 introduced `super`. At its simplest, `super` is a pointer to the current object's prototype, effectively the `Object.getPrototypeOf(this)` value. Knowing that, you can simplify the `getGreeting()` method as follows:

```
let friend = {
    getGreeting() {
        // in the previous example, this is the same as:
        // Object.getPrototypeOf(this).getGreeting.call(this)
        return super.getGreeting() + ", hi!";
    }
};
```

The call to `super.getGreeting()` is the same as `Object.getPrototypeOf(this).getGreeting.call(this)` in this context. Similarly, you can call any method on an object's prototype by using a `super` reference, as long as it's inside a concise method. Attempting to use `super` outside of concise methods results in a syntax error, as in this example:

```
let friend = {
    getGreeting: function() {
        // syntax error
        return super.getGreeting() + ", hi!";
    }
};
```

This example uses a named property with a function, and the call to `super.getGreeting()` results in a syntax error because `super` is invalid in this context.

The `super` reference is really helpful when you have multiple levels of inheritance, because in that case, `Object.getPrototypeOf()` no longer works in all circumstances. For example:

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

// prototype is person
let friend = {
    getGreeting() {
        return Object.getPrototypeOf(this).getGreeting.call(this) + ", hi!";
    }
};
Object.setPrototypeOf(friend, person);


// prototype is friend
let relative = Object.create(friend);

console.log(person.getGreeting());                  // "Hello"
console.log(friend.getGreeting());                  // "Hello, hi!"
console.log(relative.getGreeting());                // error!
```

When `relative.getGreeting()` is called, the call to `Object.getPrototypeOf()` results in an error. The reason is that `this` is `relative`, and the prototype of `relative` is the `friend` object. When `friend.getGreeting().call()` is called with `relative` as `this`, the process starts over again and continues to call recursively until a stack overflow error occurs.

This problem is difficult to solve in ECMAScript 5, but with ECMA-Script 6 and `super`, it's easy:

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

// prototype is person
let friend = {
    getGreeting() {
        return super.getGreeting() + ", hi!";
    }
};
Object.setPrototypeOf(friend, person);

// prototype is friend
let relative = Object.create(friend);

console.log(person.getGreeting());                  // "Hello"
console.log(friend.getGreeting());                  // "Hello, hi!"
console.log(relative.getGreeting());                // "Hello, hi!"
```

Because `super` references are not dynamic, they always refer to the correct object. In this case, `super.getGreeting()` always refers to `person.getGreeting()` regardless of how many other objects inherit the method.

A Formal Method Definition

Prior to ECMAScript 6, the concept of a "method" wasn't formally defined. Methods were just object properties that contained functions instead of data. ECMAScript 6 formally defines a method as a function that has an internal `[[HomeObject]]` property containing the object to which the method belongs. Consider the following:

```
let person = {

    // method
    getGreeting() {
        return "Hello";
    }
};

// not a method
function shareGreeting() {
    return "Hi!";
}
```

This code example defines `person` with a single method called `getGreeting()`. The `[[HomeObject]]` for `getGreeting()` is `person` by virtue of assigning the function directly to an object. However, the `shareGreeting()` function has no `[[HomeObject]]` specified because it wasn't assigned to an object when it was created. In most cases, this difference isn't important, but it becomes very important when using `super` references.

Any reference to `super` uses the `[[HomeObject]]` to determine what to do. The first step in the process is to call `Object.getPrototypeOf()` on the `[[HomeObject]]` to retrieve a reference to the prototype. Next, the prototype is searched for a function with the same name. Then, the `this` binding is set and the method is called. Here's an example:

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

// prototype is person
let friend = {
    getGreeting() {
        return super.getGreeting() + ", hi!";
    }
};
Object.setPrototypeOf(friend, person);

console.log(friend.getGreeting());  // "Hello, hi!"
```

Calling `friend.getGreeting()` returns a string, which combines the value from `person.getGreeting()` with `", hi!"`. The `[[HomeObject]]` of `friend.getGreeting()` is `friend`, and the prototype of `friend` is `person`, so `super.getGreeting()` is equivalent to `person.getGreeting.call(this)`.

Summary

Objects are the center of JavaScript programming, and ECMAScript 6 makes some helpful changes to objects that make them easier to work with and more flexible.

ECMAScript 6 makes several changes to object literals. Shorthand property definitions make assigning properties with the same names as in-scope variables simpler. Computed property names allow you to specify non-literal values as property names, which you've been able to do in other areas of the language. Shorthand methods let you type far fewer characters to define methods on object literals by completely omitting the colon and `function` keyword. ECMAScript 6 loosens the strict mode check for duplicate object literal property names as well, meaning two properties with the same name can be in a single object literal without throwing an error.

The `Object.assign()` method makes it easier to change multiple properties on a single object at once and is very useful when you use the mixin pattern. The `Object.is()` method performs strict equality on any value, effectively becoming a safer version of `===` when you're working with special JavaScript values.

ECMAScript 6 clearly defines enumeration order for own properties. When enumerating properties, numeric keys always come first in ascending order followed by string keys in insertion order and symbol keys in insertion order.

It's now possible to modify an object's prototype after it's been created thanks to ECMAScript 6's `Object.setPrototypeOf()` method.

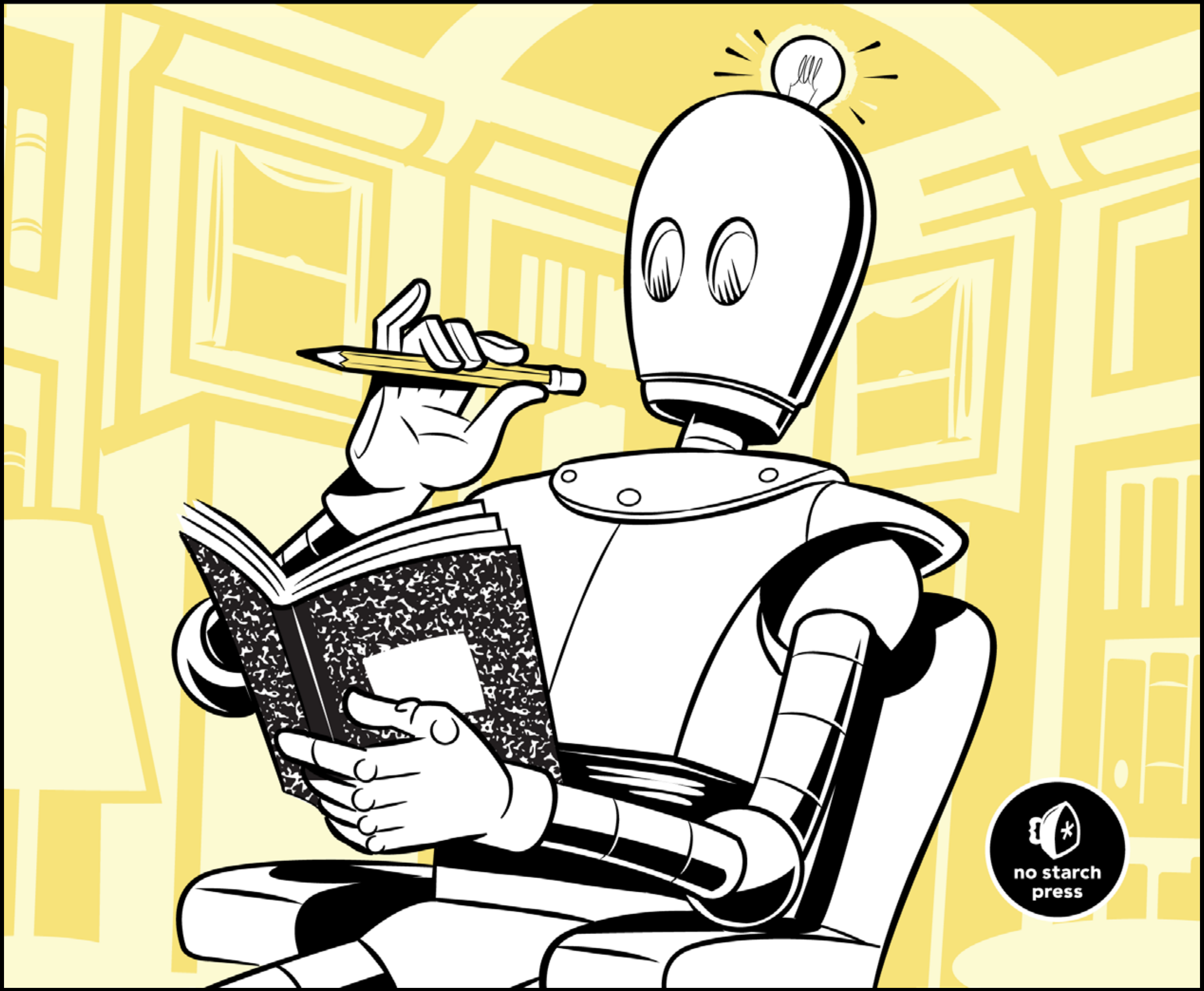
In addition, you can use the `super` keyword to call methods on an object's prototype. The `this` binding inside a method invoked using `super` is set up to automatically work with the current value of `this`.

WICKED COOL SHELL SCRIPTS

2ND EDITION

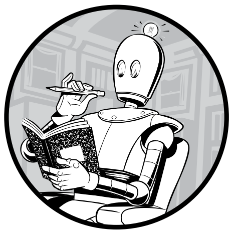
101 SCRIPTS FOR LINUX, OS X, AND UNIX SYSTEMS

DAVE TAYLOR AND BRANDON PERRY



no starch press

7
WEB AND INTERNET USERS



One area where Unix really shines is the internet. Whether you want to run a fast server from under your desk or simply surf the web intelligently and efficiently, there's little you can't embed in a shell script when it comes to internet interaction.

Internet tools are scriptable, even though you might never have thought of them that way. For example, FTP, a program that is perpetually trapped in debug mode, can be scripted in some very interesting ways, as is explored in Script #53 on page 148. Shell scripting can often improve the performance and output of most command line utilities that work with some facet of the internet.

In the first edition of this book, I (Dave) assured readers that the best tool in the internet scripter's toolbox is `lynx`; now we recommend using `curl` instead. Both tools offer a text-only interface to the web, but while `lynx` tries to offer a browser-like experience, `curl` is designed specifically for scripts, dumping out the raw HTML source of any page you'd like to examine.

For example, the following shows the top seven lines of the source from the home page of my film review blog *http://www.daveonfilm.com/*, courtesy of `curl`:

```
$ curl -s http://www.daveonfilm.com/ | head -7
<!DOCTYPE html>
<html lang="en-US">
<head>
<meta charset="UTF-8" />
<link rel="profile" href="http://gmpg.org/xfn/11" />
<link rel="pingback" href="http://www.daveonfilm.com/xmlrpc.php" />
<title>Dave On Film: Smart Movie Reviews from Dave Taylor</title>
```

You can accomplish the same result with `lynx` if `curl` isn't available, but if you have both, we recommend `curl`. That's what we'll work with in this chapter.

WARNING *One limitation to the website scraper scripts in this chapter is that if the script depends on a website that's changed its layout or API in the time since this book was written, the script might be broken. But if you can read HTML or JSON (even if you don't understand it all), you should be able to fix any of these scripts. The problem of tracking other sites is exactly why Extensible Markup Language (XML) was created: it allows site developers to provide the content of a web page separately from the rules for its layout.*

#53 Downloading Files via FTP

One of the original killer apps of the internet was file transfer, and one of the simplest solutions is FTP, File Transfer Protocol. At a fundamental level, all internet interaction is based on file transfer, whether it's a web browser requesting an HTML document and its accompanying image files, a chat server relaying lines of discussion back and forth, or an email message traveling from one end of the earth to the other.

The original FTP program still lingers on, and while its interface is crude, the program is powerful, capable, and well worth taking advantage of. There are plenty of newer FTP programs around, notably FileZilla (*http://filezilla-project.org/*) or NcFTP (*http://www.ncftp.org/*), plus lots of nice graphical interfaces you can add to FTP to make it more user-friendly. With the help of some shell script wrappers, however, FTP does just fine for uploading and downloading files.

For example, a typical use case for FTP is to download files from the internet, which we'll do with the script in Listing 7-1. Quite often, the files will be located on anonymous FTP servers and will have URLs similar to *ftp://<someserver>/<path>/<filename>/*.

The Code

```
#!/bin/bash

# ftpget--Given an ftp-style URL, unwraps it and tries to obtain the
#   file using anonymous ftp.

anonpass="$LOGNAME@$(hostname)"

if [ $# -ne 1 ] ; then
  echo "Usage: $0 ftp://..." >&2
  exit 1
fi

# Typical URL: ftp://ftp.ncftp.com/unixstuff/q2getty.tar.gz

if [ "$(echo $1 | cut -c1-6)" != "ftp://" ] ; then
  echo "$0: Malformed url. I need it to start with ftp://" >&2;
  exit 1
fi

server="$(echo $1 | cut -d/ -f3)"
filename="$(echo $1 | cut -d/ -f4-)"
basefile="$(basename $filename)"

echo ${0}: Downloading $basefile from server $server

❶ ftp -np << EOF
open $server
user ftp $anonpass
get "$filename" "$basefile"
quit
EOF

if [ $? -eq 0 ] ; then
  ls -l $basefile
fi

exit 0
```

Listing 7-1: The ftpget script

How It Works

The heart of this script is the sequence of commands fed to the FTP program starting at ❶. This illustrates the essence of a batch file: a sequence of instructions that's fed to a separate program so that the receiving program (in this case FTP) thinks the instructions are being entered by the user. Here we specify the server connection to open, specify the anonymous user

(FTP) and whatever default password is specified in the script configuration (typically your email address), and then get the specified file from the FTP site and quit the transfer.

Running the Script

This script is straightforward to use: just fully specify an FTP URL, and it'll download the file to the current working directory, as Listing 7-2 details.

The Results

```
$ ftpget ftp://ftp.ncftp.com/unixstuff/q2getty.tar.gz
ftpget: Downloading q2getty.tar.gz from server ftp.ncftp.com
-rw-r--r--  1 taylor  staff  4817 Aug 14  1998 q2getty.tar.gz
```

Listing 7-2: Running the ftpget script

Some versions of FTP are more verbose than others, and because it's not too uncommon to find a slight mismatch in the client and server protocol, those verbose versions of FTP can spit out scary-looking errors, like `Unimplemented command`. You can safely ignore these. For example, Listing 7-3 shows the same script run on OS X.

```
$ ftpget ftp://ftp.ncftp.com/ncftp/ncftp-3.1.5-src.tar.bz2
../Scripts.new/053-ftpget.sh: Downloading q2getty.tar.gz from server ftp.
ncftp.com
Connected to ncftp.com.
220 ncftpd.com NcFTPd Server (licensed copy) ready.
331 Guest login ok, send your complete e-mail address as password.
230-You are user #2 of 16 simultaneous users allowed.
230-
230 Logged in anonymously.
Remote system type is UNIX.
Using binary mode to transfer files.
local: q2getty.tar.gz remote: unixstuff/q2getty.tar.gz
227 Entering Passive Mode (209,197,102,38,194,11)
150 Data connection accepted from 97.124.161.251:57849; transfer starting for
q2getty.tar.gz (4817 bytes).
100% |***************************************************|  4817
67.41 KiB/s    00:00 ETA
226 Transfer completed.
4817 bytes received in 00:00 (63.28 KiB/s)
221 Goodbye.
-rw-r--r--  1 taylor  staff  4817 Aug 14  1998 q2getty.tar.gz
```

Listing 7-3: Running the ftpget script on OS X

If your FTP is excessively verbose and you're on OS X, you can quiet it down by adding a `-V` flag to the FTP invocation in the script (that is, instead of FTP `-n`, use FTP `-nV`).

Hacking the Script

This script can be expanded to decompress the downloaded file automatically (see Script #33 on page 101 for an example of how to do this) if it has certain file extensions. Many compressed files such as *.tar.gz* and *.tar.bz2* can be decompressed by default with the system `tar` command.

You can also tweak this script to make it a simple tool for *uploading* a specified file to an FTP server. If the server supports anonymous connections (few do nowadays, thanks to script kiddies and other delinquents, but that's another story), all you really have to do is specify a destination directory on the command line or in the script and change the `get` to a `put` in the main script, as shown here:

```
ftp -np << EOF

open $server

user ftp $anonpass

cd $destdir

put "$filename"

quit
EOF
```

To work with a password-protected account, you could have the script prompt for the password interactively by turning off echoing before a `read` statement and then turning it back on when you're done:

```
/bin/echo -n "Password for ${user}: "

stty -echo

read password

stty echo
echo ""
```

A smarter way to prompt for a password, however, is to just let the FTP program do the work itself. This will happen as written in our script because if a password is required to gain access to the specified FTP account, the FTP program itself will prompt for it.

#54 Extracting URLs from a Web Page

A straightforward shell script application of `lynx` is to extract a list of URLs on a given web page, which can be quite helpful when scraping the internet for links. We said we'd switched from `lynx` to `curl` for this edition of the book, but it turns out that `lynx` is about a hundred times easier to use for this script (see Listing 7-4) than `curl`, because `lynx` parses HTML automatically whereas `curl` forces you to parse the HTML yourself.

Don't have `lynx` on your system? Most Unix systems today have package managers such as `yum` on Red Hat, `apt` on Debian, and `brew` on OS X (though `brew` is not installed by default) that you can use to install `lynx`. If you prefer to compile `lynx` yourself, or just want to download prebuilt binaries, you can download it from *http://lynx.browser.org/*.

The Code

```
#!/bin/bash

# getlinks--Given a URL, returns all of its relative and absolute links.
#   Has three options: -d to generate the primary domains of every link,
#   -i to list just those links that are internal to the site (that is,
#   other pages on the same site), and -x to produce external links only
#   (the opposite of -i).

if [ $# -eq 0 ] ; then
  echo "Usage: $0 [-d|-i|-x] url"  >&2
  echo "-d=domains only, -i=internal refs only, -x=external only" >&2
  exit 1
fi

if [ $# -gt 1 ] ; then
  case "$1" in
❶    -d) lastcmd="cut -d/ -f3|sort|uniq"
        shift
        ;;
    -r) basedomain="http://$(echo $2 | cut -d/ -f3)/"
❷        lastcmd="grep \"^$basedomain\"|sed \"s|$basedomain||g\"|sort|uniq"
        shift
        ;;
    -a) basedomain="http://$(echo $2 | cut -d/ -f3)/"
❸        lastcmd="grep -v \"^$basedomain\"|sort|uniq"
        shift
        ;;
     *) echo "$0: unknown option specified: $1" >&2; exit 1
  esac
else
❹  lastcmd="sort|uniq"
fi

lynx -dump "$1"|\
❺  sed -n '/^References$/,$p'|\
  grep -E '[[:digit:]]+\.'|\
  awk '{print $2}'|\
      cut -d\? -f1|\
❻     eval $lastcmd

exit 0
```

Listing 7-4: The `getlinks` script

How It Works

When displaying a page, `lynx` shows the text of the page formatted as best it can followed by a list of all hypertext references, or links, found on that page. This script extracts just the links by using a `sed` invocation to print everything after the `"References"` string in the web page text ❺. Then the script processes the list of links as needed based on the user-specified flags.

One interesting technique demonstrated by this script is the way the variable `lastcmd` (❶, ❷, ❸, ❹) is set to filter the list of links that it extracts according to the flags specified by the user. Once `lastcmd` is set, the amazingly handy `eval` command ❻ is used to force the shell to interpret the content of the variable as if it were a command instead of a variable.

Running the Script

By default, this script outputs a list of all links found on the specified web page, not just those that are prefaced with `http:`. There are three optional command flags that can be specified to change the results, however: `-d` produces just the domain names of all matching URLs, `-r` produces a list of just the *relative* references (that is, those references that are found on the same server as the current page), and `-a` produces just the *absolute* references (those URLs that point to a different server).

The Results

A simple request is a list of all links on a specified website home page, as Listing 7-5 shows.

```
$ getlinks http://www.daveonfilm.com/ | head -10
http://instagram.com/d1taylor
http://pinterest.com/d1taylor/
http://plus.google.com/110193533410016731852
https://plus.google.com/u/0/110193533410016731852
https://twitter.com/DaveTaylor
http://www.amazon.com/Doctor-Who-Shada-Adventures-Douglas/
http://www.daveonfilm.com/
http://www.daveonfilm.com/about-me/
http://www.daveonfilm.com/author/d1taylor/
http://www.daveonfilm.com/category/film-movie-reviews/
```

Listing 7-5: Running the `getlinks` script

Another possibility is to request a list of all domain names referenced at a specific site. This time, let's first use the standard Unix tool wc to check how many links are found overall:

```
$ getlinks http://www.amazon.com/ | wc -l
     219
```

Amazon has 219 links on its home page. Impressive! How many different domains does that represent? Let's generate a list with the -d flag:

```
$ getlinks -d http://www.amazon.com/ | head -10
amazonlocal.com
aws.amazon.com
fresh.amazon.com
kdp.amazon.com
services.amazon.com
www.6pm.com
www.abebooks.com
www.acx.com
www.afterschool.com
www.alexa.com
```

Amazon doesn't tend to point outside its own site, but there are some partner links that creep onto the home page. Other sites are different, of course.

What if we split the links on the Amazon page into relative and absolute links?

```
$ getlinks -a http://www.amazon.com/ | wc -l
51
$ getlinks -r http://www.amazon.com/ | wc -l
222
```

As I expected, Amazon has four times more relative links pointing inside its own site than it has absolute links, which would lead to a different website. Gotta keep those customers on your own page!

Hacking the Script

You can see where getlinks could be quite useful as a site analysis tool. For a way to enhance the script, stay tuned: Script #69 on page 209 complements this script nicely, allowing us to quickly check that all hypertext references on a site are valid.

#55 Getting GitHub User Information

GitHub has grown to be a huge boon to the open source industry and open collaboration across the world. Many system administrators and developers have visited GitHub to pull down some source code or report an issue to an open source project. Because GitHub is essentially a social platform for developers, getting to know a user's basic information quickly can be useful. The script in Listing 7-6 prints some information about a given GitHub user, and it gives a good introduction to the very powerful GitHub API.

The Code

```
#!/bin/bash
# githubuser--Given a GitHub username, pulls information about them.

if [ $# -ne 1 ]; then
  echo "Usage: $0 <username>"
  exit 1
fi

# The -s silences curl's normally verbose output.
❶ curl -s "https://api.github.com/users/$1" | \
        awk -F'"' '
            /\"name\":/ {
              print $4" is the name of the Github user."
            }
            /\"followers\":/{
              split($3, a, " ")
              sub(/,/, "", a[2])
              print "They have "a[2]" followers."
            }
            /\"following\":/{
              split($3, a, " ")
              sub(/,/, "", a[2])
              print "They are following "a[2]" other users."
            }
            /\"created_at\":/{
              print "Their account was created on "$4"."
            }
            '
exit 0
```

Listing 7-6: The `githubuser` *script*

How It Works

I'll admit, this is almost more of an `awk` script than a Bash script, but sometimes you need the extra horsepower `awk` provides for parsing (the GitHub API returns JSON). We use `curl` to ask GitHub for the user ❶, given as the argument of the script, and pipe the JSON to `awk`. With `awk`, we specify a field separator of the double quotes character, as this will make parsing the JSON much simpler. Then we match the JSON with a handful of regular expressions in the `awk` script and print the results in a user-friendly way.

Running the Script

The script accepts a single argument: the user to look up on GitHub. If the username provided doesn't exist, nothing will be printed.

The Results

When passed a valid username, the script should print a user-friendly summary of the GitHub user, as Listing 7-7 shows.

```
$ githubuser brandonprry
Brandon Perry is the name of the Github user.
They have 67 followers.
They are following 0 other users.
Their account was created on 2010-11-16T02:06:41Z.
```

Listing 7-7: Running the `githubuser` script

Hacking the Script

This script has a lot of potential due to the information that can be retrieved from the GitHub API. In this script, we are only printing four values from the JSON returned. Generating a "résumé" for a given user based on the information provided by the API, like those provided by many web services, is just one possibility.

#56 ZIP Code Lookup

To demonstrate a different technique for scraping the web, this time using `curl`, let's create a simple ZIP code lookup tool. Give the script in Listing 7-8 a ZIP code, and it'll report the city and state the code belongs to. Easy enough.

Your first instinct might be to use the official US Postal Service website, but we're going to tap into a different site, *http://city-data.com/*, which configures each ZIP code as its own web page so information is far easier to extract.

The Code

```
#!/bin/bash

# zipcode--Given a ZIP code, identifies the city and state. Use city-data.com,
#   which has every ZIP code configured as its own web page.

baseURL="http://www.city-data.com/zips"

/bin/echo -n "ZIP code $1 is in "

curl -s -dump "$baseURL/$1.html" | \
   grep -i '<title>' | \
   cut -d\( -f2 | cut -d\) -f1

exit 0
```

Listing 7-8: The `zipcode` script

How It Works

The URLs for ZIP code information pages on *http://city-data.com/* are structured consistently, with the ZIP code itself as the final part of the URL.

```
http://www.city-data.com/zips/80304.html
```

This consistency makes it quite easy to create an appropriate URL for a given ZIP code on the fly. The resultant page has the city name in the title, conveniently denoted by open and close parentheses. So the page for the previous example has this as its title:

```
<title>80304 Zip Code (Boulder, Colorado) Profile - homes, apartments,
schools, population, income, averages, housing, demographics, location,
statistics, residents and real estate info</title>
```

Long, but pretty easy to work with!

Running the Script

The standard way to invoke the script is to specify the desired ZIP code on the command line. If it's valid, the city and state will be displayed, as shown in Listing 7-9.

The Results

```
$ zipcode 10010
ZIP code 10010 is in New York, New York
$ zipcode 30001
ZIP code 30001 is in <title>Page not found – City-Data.com</title>
$ zipcode 50111
ZIP code 50111 is in Grimes, Iowa
```

Listing 7-9: Running the `zipcode` script

Since 30001 isn't a real ZIP code, the script generates a `Page not found` error. That's a bit sloppy, and we can do better.

Hacking the Script

The most obvious hack to this script would be to do something in response to errors other than just spew out that ugly `<title>Page not found – City-Data.com</title>` sequence. More useful still would be to add a `-a` flag that tells the script to display more information about the specified region, since *http://city-data.com/* offers quite a bit of information beyond city names—including land area, population demographics, and home prices.

#57 Area Code Lookup

A variation on the theme of the ZIP code lookup in Script #56 is an area code lookup. This one turns out to be really simple, because there are some very easy-to-parse web pages with area codes. The page at *http://www.bennetyee.org/ucsd-pages/area.html* is particularly easy to parse, not only because it is in tabular form but also because the author has identified elements with HTML attributes. For example, the line that defines area code 207 reads like so:

```
<tr><td align=center><a name="207">207</a></td><td align=center>ME</td><td
align=center>-5</td><td>   Maine</td></tr>
```

We'll use this site to look up area codes in the script in Listing 7-10.

The Code

```
#!/bin/bash

# areacode--Given a three-digit US telephone area code, identifies the city
#   and state using the simple tabular data at Bennet Yee's website.

source="http://www.bennetyee.org/ucsd-pages/area.html"

if [ -z "$1" ] ; then
  echo "usage: areacode <three-digit US telephone area code>"; exit 1
fi

# wc -c returns characters + end of line char, so 3 digits = 4 chars
if [ "$(echo $1 | wc -c)" -ne 4 ] ; then
  echo "areacode: wrong length: only works with three-digit US area codes"
  exit 1
fi

# Are they all digits?
if [ ! -z "$(echo $1 | sed 's/[[:digit:]]//g')" ] ; then
  echo "areacode: not-digits: area codes can only be made up of digits";
exit 1
fi

# Now, finally, let's look up the area code...

result="$(❶curl -s -dump $source | grep "name=\"$1" | \
  sed 's/<[^>]*>//g;s/^ //g' | \
  cut -f2- -d\ | cut -f1 -d\( )"

echo "Area code $1 =$result"

exit 0
```

Listing 7-10: The areacode script

How It Works

The code in this shell script is mainly input validation, ensuring the data provided by the user is a valid area code. The core of the script is a `curl` call ❶, whose output is piped to `sed` for cleaning up and then trimmed with `cut` to what we want to display to the user.

Running the Script

This script takes a single argument, the area code to look up information for. Listing 7-11 gives examples of the script in use.

The Results

```
$ areacode 817
Area code 817 =  N Cent. Texas: Fort Worth area
$ areacode 512
Area code 512 =  S Texas: Austin
$ areacode 903
Area code 903 =  NE Texas: Tyler
```

Listing 7-11: Testing the `areacode` *script*

Hacking the Script

A simple hack would be to invert the search so that you provide a state and city and the script prints all of the area codes for the given city.

#58 Keeping Track of the Weather

Being inside an office or server room with your nose to a terminal all day sometimes makes you yearn to be outside, especially when the weather is really nice. *http://www.wunderground.com/* is a great website, and it actually offers a free API for developers if you sign up for an API key. With the API key, we can write a quick shell script (shown in Listing 7-12) to tell us just how nice (or poor) the weather is outside. Then we can decide whether taking a quick walk is really a good idea.

The Code

```
#!/bin/bash
# weather--Gets the weather for a specific region or ZIP code.

if [ $# -ne 1 ]; then
  echo "Usage: $0 <zipcode>"
  exit 1
fi

apikey="b03fdsaf3b2e7cd23" # Not a real API key--you need your own.
```

```
❶ weather=`curl -s \
      "https://api.wunderground.com/api/$apikey/conditions/q/$1.xml"`
❷ state=`xmllint --xpath \
       //response/current_observation/display_location/full/text\(\) \
       <(echo $weather)`
  zip=`xmllint --xpath \
       //response/current_observation/display_location/zip/text\(\) \
       <(echo $weather)`
  current=`xmllint --xpath \
       //response/current_observation/temp_f/text\(\) \
       <(echo $weather)`
  condition=`xmllint --xpath \
       //response/current_observation/weather/text\(\) \
       <(echo $weather)`

  echo $state" ("$zip") : Current temp "$current"F and "$condition" outside."

  exit 0
```

Listing 7-12: The weather *script*

How It Works

In this script, we use `curl` to call the Wunderground API and save the HTTP response data in the `weather` variable ❶. We then use the `xmllint` (easily installable with your favorite package manager such as `apt`, `yum`, or `brew`) utility to perform an XPath query on the data returned ❷. We also use an interesting syntax in Bash when calling `xmllint` with the `<(echo $weather)` at the end. This syntax takes the output of the inner command and passes it to the command as a file descriptor, so the program thinks it's reading a real file. After gathering all the relevant information from the XML returned, we print a friendly message with general weather stats.

Running the Script

When you invoke the script, just specify the desired ZIP code, as Listing 7-13 shows. Easy enough!

The Results

```
$ weather 78727
Austin, TX (78727) : Current temp 59.0F and Clear outside.
$ weather 80304
Boulder, CO (80304) : Current temp 59.2F and Clear outside.
$ weather 10010
New York, NY (10010) : Current temp 68.7F and Clear outside.
```

Listing 7-13: Testing the weather *script*

Hacking the Script

We have a secret. This script can actually take more than just ZIP codes. You can also specify regions in the Wunderground API, such as `CA/San_Francisco` (try it as an argument to the weather script!). However, this format isn't incredibly user-friendly: it requires underscores instead of spaces and the slash in the middle. Adding the ability to ask for the state abbreviation and the city and then replacing any spaces with underscores if no arguments are passed would be a useful addition. As usual, this script could do with more error-checking code. What happens if you enter a four-digit ZIP code? Or a ZIP code that's not assigned?

#59 Digging Up Movie Info from IMDb

The script in Listing 7-14 demonstrates a more sophisticated way to access the internet through `lynx`, by searching the Internet Movie Database (*http://www.imdb.com/*) to find films that match a specified pattern. IMDb assigns every movie, TV series, and even TV episode a unique numeric code; if the user specifies that code, this script will return a synopsis of the film. Otherwise, it will return a list of matching films from a title or partial title.

The script accesses different URLs depending on the type of query (numeric ID or file title) and then caches the results so it can dig through the page multiple times to extract different pieces of information. And it uses a lot—a *lot*!—of calls to `sed` and `grep`, as you'll see.

The Code

```
#!/bin/bash
# moviedata--Given a movie or TV title, returns a list of matches. If the user
#   specifies an IMDb numeric index number, however, returns the synopsis of
#   the film instead. Uses the Internet Movie Database.

titleurl="http://www.imdb.com/title/tt"
imdburl="http://www.imdb.com/find?s=tt&exact=true&ref_=fn_tt_ex&q="
tempout="/tmp/moviedata.$$"

❶ summarize_film()
{
   # Produce an attractive synopsis of the film.

   grep "<title>" $tempout | sed 's/<[^>]*>//g;s/(more)//'

   grep --color=never -A2 '<h5>Plot:' $tempout | tail -1 | \
     cut -d\< -f1 | fmt | sed 's/^/    /'

   exit 0
}
```

```
  trap "rm -f $tempout" 0 1 15

  if [ $# -eq 0 ] ; then
    echo "Usage: $0 {movie title | movie ID}" >&2
    exit 1
  fi

  #########
  # Checks whether we're asking for a title by IMDb title number.

  nodigits="$(echo $1 | sed 's/[[:digit:]]*//g')"

  if [ $# -eq 1 -a -z "$nodigits" ] ; then
    lynx -source "$titleurl$1/combined" > $tempout
    summarize_film
    exit 0
  fi

  ##########
  # It's not an IMDb title number, so let's go with the search...

  fixedname="$(echo $@ | tr ' ' '+')"       # for the URL

  url="$imdburl$fixedname"

❷ lynx -source $imdburl$fixedname > $tempout

  # No results?

❸ fail="$(grep --color=never '<h1 class="findHeader">No ' $tempout)"

  # If there's more than one matching title...

  if [ ! -z "$fail" ] ; then
    echo "Failed: no results found for $1"
    exit 1
  elif [ ! -z "$(grep '<h1 class="findHeader">Displaying' $tempout)" ] ; then
    grep --color=never '/title/tt' $tempout | \
    sed 's/</\
  </g' | \
    grep -vE '(.png|.jpg|>[ ]*$)' | \
    grep -A 1 "a href=" | \
    grep -v '^--$' | \
    sed 's/<a href="\/title\/tt//g;s/<\/a> //' | \
❹   awk '(NR % 2 == 1) { title=$0 } (NR % 2 == 0) { print title " " $0 }' | \
    sed 's/\/.*>/: /' | \
    sort
  fi


  exit 0
```

Listing 7-14: The moviedata *script*

How It Works

This script builds a different URL depending on whether the command argument specified is a film title or an IMDb ID number. If the user specifies a title by ID number, the script builds the appropriate URL, downloads it, saves the `lynx` output to the `$tempout` file ❷, and finally calls `summarize_film()` ❶. Not too difficult.

But if the user specifies a title, then the script builds a URL for a search query on IMDb and saves the results page to the temp file. If IMDb can't find a match, then the `<h1>` tag with `class="findHeader"` value in the returned HTML will say `No results`. That's what the invocation at ❸ checks. Then the test is easy: if `$fail` is not zero length, the script can report that no results were found.

If the result *is* zero length, however, that means that `$tempfile` now contains one or more successful search results for the user's pattern. These results can all be extracted by searching for `/title/tt` as a pattern within the source, but there's a caveat: IMDb doesn't make it easy to parse the results because there are multiple matches to any given title link. The rest of that gnarly `sed|grep|sed` sequence tries to identify and remove the duplicate matches, while still retaining the ones that matter.

Further, when IMDb has a match like `"Lawrence of Arabia (1962)"`, it turns out that the title and year are two different HTML elements on two different lines in the result. Ugh. We need the year, however, to differentiate films with the same title that were released in different years. That's what the `awk` statement at ❹ does, in a tricky sort of way.

If you're unfamiliar with `awk`, the general format for an `awk` script is `(`*`condition`*`) {` *`action`* `}`. This line saves odd-numbered lines in `$title` and then, on even-numbered lines (the year and match type data), it outputs both the previous and the current line's data as one line of output.

Running the Script

Though short, this script is quite flexible with input formats, as can be seen in Listing 7-15. You can specify a film title in quotes or as separate words, and you can then specify the eight-digit IMDb ID value to select a specific match.

The Results

```
$ moviedata lawrence of arabia
0056172: Lawrence of Arabia (1962)
0245226: Lawrence of Arabia (1935)
0390742: Mighty Moments from World History (1985) (TV Series)
1471868: Mystery Files (2010) (TV Series)
1471868: Mystery Files (2010) (TV Series)
1478071: Lawrence of Arabia (1985) (TV Episode)
1942509: Lawrence of Arabia (TV Episode)
1952822: Lawrence of Arabia (2011) (TV Episode)
```

```
$ moviedata 0056172
Lawrence of Arabia (1962)
    A flamboyant and controversial British military figure and his
    conflicted loyalties during his World War I service in the Middle East.
```

Listing 7-15: Running the `moviedata` script

Hacking the Script

The most obvious hack to this script would be to get rid of the ugly IMDb movie ID numbers in the output. It would be straightforward to hide the movie IDs (because the IDs as shown are rather unfriendly and prone to mistyping) and have the shell script output a simple menu with unique index values that can then be typed in to select a particular film.

In situations where there's exactly one film matched (try `moviedata monsoon wedding`), it would be great for the script to recognize that it's the only match, grab the movie number for the film, and reinvoke itself to get that data. Give it a whirl!

A problem with this script, as with most scripts that scrape values from a third-party website, is that if IMDb changes its page layout, the script will break and you'll need to rebuild the script sequence. It's a lurking bug but, with a site like IMDb that hasn't changed in years, probably not a dangerous one.

#60 Calculating Currency Values

In the first edition of this book, currency conversion was a remarkably difficult task requiring two scripts: one to pull conversion rates from a financial website and save them in a special format and another to use that data to actually do the conversion—say from US dollars to Euros. In the intervening years, however, the web has become quite a bit more sophisticated, and there's no reason for us to go through tons of work when sites like Google offer simple, script-friendly calculators.

For this version of the currency conversion script, shown in Listing 7-16, we're just going to tap into the currency calculator at *http://www.google.com/finance/converter*.

The Code

```
#!/bin/bash

# convertcurrency--Given an amount and base currency, converts it to the
#   specified target currency using ISO currency identifiers.
#   Uses Google's finance converter for the heavy lifting:
#   http://www.google.com/finance/converter
```

```
if [ $# -eq 0 ]; then
   echo "Usage: $(basename $0) amount currency to currency"
   echo "Most common currencies are CAD, CNY, EUR, USD, INR, JPY, and MXN"
   echo "Use \"$(basename $0) list\" for the full list of supported
currencies."
fi

if [ $(uname) = "Darwin" ]; then
  LANG=C # For an issue on OS X with invalid byte sequences and lynx
fi
     url="https://www.google.com/finance/converter"
tempfile="/tmp/converter.$$"
    lynx=$(which lynx)

# Since this has multiple uses, let's grab this data before anything else.

currencies=$($lynx -source "$url" | grep "option  value=" | \
    cut -d\" -f2- | sed 's/">/ /' | cut -d\( -f1 | sort | uniq)

########### Deal with all non-conversion requests.

if [ $# -ne 4 ] ; then
  if [ "$1" = "list" ] ; then
    # Produce a listing of all currency symbols known by the converter.
    echo "List of supported currencies:"
    echo "$currencies"
  fi
  exit 0
fi

########### Now let's do a conversion.

if [ $3 != "to" ] ; then
  echo "Usage: $(basename $0) value currency TO currency"
  echo "(use \"$(basename $0) list\" to get a list of all currency values)"
  exit 0
fi

amount=$1
basecurrency="$(echo $2 | tr '[:lower:]' '[:upper:]')"
targetcurrency="$(echo $4 | tr '[:lower:]' '[:upper:]')"

# And let's do it--finally!

$lynx -source "$url?a=$amount&from=$basecurrency&to=$targetcurrency" | \
  grep 'id=currency_converter_result' | sed 's/<[^>]*>//g'

exit 0
```

Listing 7-16: The convertcurrency script

How It Works

The Google Currency Converter has three parameters that are passed via the URL itself: the amount, the original currency, and the currency you want to convert to. You can see this in action in the following request to convert 100 US dollars into Mexican pesos.

```
https://www.google.com/finance/converter?a=100&from=USD&to=MXN
```

In the most basic use case, then, the script expects the user to specify each of those three fields as arguments, and then passes it all to Google in the URL.

The script also has some usage messages that make it a lot easier to use. To see those, let's just jump to the demonstration portion, shall we?

Running the Script

This script is designed to be easy to use, as Listing 7-17 details, though a basic knowledge of at least a few countries' currencies is beneficial.

The Results

```
$ convertcurrency
Usage: convert amount currency to currency
Most common currencies are CAD, CNY, EUR, USD, INR, JPY, and MXN
Use "convertcurrency list" for the full list of supported currencies.
$ convertcurrency list | head -10
List of supported currencies:

AED United Arab Emirates Dirham
AFN Afghan Afghani
ALL Albanian Lek
AMD Armenian Dram
ANG Netherlands Antillean Guilder
AOA Angolan Kwanza
ARS Argentine Peso
AUD Australian Dollar
AWG Aruban Florin
$ convertcurrency 75 eur to usd
75 EUR = 84.5132 USD
```

Listing 7-17: Running the convertcurrency script

Hacking the Script

While this web-based calculator is austere and simple to work with, the output could do with some cleaning up. For example, the output in Listing 7-17 doesn't entirely make sense because it expresses US dollars with four digits after the decimal point, even though cents only go to two digits. The correct output should be 84.51, or if rounded up, 84.52. That's something fixable in the script.

While you're at it, validating currency abbreviations would be beneficial. And in a similar vein, changing those abbreviated currency codes to proper currency names would be a nice feature, too, so you'd know that AWG is the Aruban florin or that BTC is Bitcoin.

#61 Retrieving Bitcoin Address Information

Bitcoin has taken the world by storm, with whole businesses built around the technology of the *blockchain* (which is the core of how Bitcoin works). For anyone who works with Bitcoin at all, getting useful information about specific Bitcoin addresses can be a major hassle. However, we can easily automate data gathering using a quick shell script, like that in Listing 7-18.

The Code

```
#!/bin/bash
# getbtcaddr--Given a Bitcoin address, reports useful information.

if [ $# -ne 1 ]; then
  echo "Usage: $0 <address>"
  exit 1
fi

base_url="https://blockchain.info/q/"

balance=`$(curl -s $base_url"addressbalance/"$1`)
recv=`$(curl -s $base_url"getreceivedbyaddress/"$1`)
sent=`$(curl -s $base_url"getsentbyaddress/"$1`)
first_made=`$(curl -s $base_url"addressfirstseen/"$1`)

echo "Details for address $1"
echo -e "\tFirst seen: "`date -d @$first_made`
echo -e "\tCurrent balance: "$balance
echo -e "\tSatoshis sent: "$sent
echo -e "\tSatoshis recv: "$recv
```

Listing 7-18: The `getbtcaddr` *script*

How It Works

This script automates a handful of `curl` calls to retrieve a few key pieces of information about a given Bitcoin address. The API available on *http://blockchain.info/* gives us very easy access to all kinds of Bitcoin and blockchain information. In fact, we don't even need to parse the responses coming back from the API, because it returns only single, simple values. After making calls to retrieve the given address's balance, how many BTC have been sent and received by it, and when it was made, we print the information to the screen for the user.

Running the Script

The script accepts only a single argument, the Bitcoin address we want information about. However, I should mention that a string passed in that is not a real Bitcoin address will simply print all 0s for the sent, received, and current balance values, as well as a creation date in the year 1969. Any nonzero values are in a unit called *satoshis*, which is the smallest denomination of a Bitcoin (like pennies, but to many more decimal places).

The Results

Running the `getbtcaddr` shell script is simple as it only takes a single argument, the Bitcoin address to request data about, as Listing 7-19 shows.

```
$ getbtcaddr 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa
Details for address 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa
    First seen: Sat Jan 3 12:15:05 CST 2009
    Current balance: 6554034549
    Satoshis sent: 0
    Satoshis recv: 6554034549
$ getbtcaddr 1EzwoHtiXB4iFwedPr49iywjZn2nnekhoj
Details for address 1EzwoHtiXB4iFwedPr49iywjZn2nnekhoj
    First seen: Sun Mar 11 11:11:41 CDT 2012
    Current balance: 2000000
    Satoshis sent: 716369585974
    Satoshis recv: 716371585974
```

Listing 7-19: Running the `getbtcaddr` *script*

Hacking the Script

The numbers printed to the screen by default are pretty large and a bit difficult for most people to comprehend. The `scriptbc` script (Script #9 on page 26) can easily be used to report in more reasonable units, such as whole Bitcoins. Adding a scale argument to the script would be an easy way for the user to get a more readable printout.

#62 Tracking Changes on Web Pages

Sometimes great inspiration comes from seeing an existing business and saying to yourself, "That doesn't seem too hard." The task of tracking changes on a website is a surprisingly simple way of collecting such inspirational material. The script in Listing 7-20, `changetrack`, automates that task. This script has one interesting nuance: when it detects changes to the site, it emails the new web page to the user, rather than just reporting the information on the command line.

The Code

```
#!/bin/bash

# changetrack--Tracks a given URL and, if it's changed since the last visit,
#   emails the new page to the specified address.

sendmail=$(which sendmail)
sitearchive="/tmp/changetrack"
tmpchanges="$sitearchive/changes.$$"  # Temp file
fromaddr="webscraper@intuitive.com"
dirperm=755        # read+write+execute for dir owner
fileperm=644       # read+write for owner, read only for others

trap "$(which rm) -f $tmpchanges" 0 1 15  # Remove temp file on exit

if [ $# -ne 2 ] ; then
  echo "Usage: $(basename $0) url email" >&2
  echo "  tip: to have changes displayed on screen, use email addr '-'" >&2
  exit 1
fi

if [ ! -d $sitearchive ] ; then
  if ! mkdir $sitearchive ; then
    echo "$(basename $0) failed: couldn't create $sitearchive." >&2
    exit 1
  fi
  chmod $dirperm $sitearchive
fi

if [ "$(echo $1 | cut -c1-5)" != "http:" ] ; then
  echo "Please use fully qualified URLs (e.g. start with 'http://')" >&2
  exit 1
fi

fname="$(echo $1 | sed 's/http:\/\///g' | tr '/?&' '...')"
baseurl="$(echo $1 | cut -d/ -f1-3)/"

# Grab a copy of the web page and put it in an archive file. Note that we
#   can track changes by looking just at the content (that is, -dump, not
#   -source), so we can skip any HTML parsing....

lynx  -dump "$1" | uniq > $sitearchive/${fname}.new
if [ -f "$sitearchive/$fname" ] ; then
  # We've seen this site before, so compare the two with diff.
  diff $sitearchive/$fname $sitearchive/${fname}.new > $tmpchanges
  if [ -s $tmpchanges ] ; then
    echo "Status: Site $1 has changed since our last check."
  else
    echo "Status: No changes for site $1 since last check"
    rm -f $sitearchive/${fname}.new     # Nothing new...
    exit 0                              # No change--we're outta here.
  fi
else
  echo "Status: first visit to $1. Copy archived for future analysis."
  mv $sitearchive/${fname}.new $sitearchive/$fname
  chmod $fileperm $sitearchive/$fname
  exit 0
fi

# If we're here, the site has changed, and we need to send the contents
#   of the .new file to the user and replace the original with the .new
#   for the next invocation of the script.

if [ "$2" != "-" ] ; then

( echo "Content-type: text/html"
  echo "From: $fromaddr (Web Site Change Tracker)"
  echo "Subject: Web Site $1 Has Changed"
❶ echo "To: $2"
  echo ""

❷ lynx -s -dump $1 | \
❸ sed -e "s|src=\"|SRC=\"$baseurl|gi" \
❹     -e "s|href=\"|HREF=\"$baseurl|gi" \
❺     -e "s|$baseurl\/http:|http:|g"
) | $sendmail -t

else
  # Just showing the differences on the screen is ugly. Solution?

  diff $sitearchive/$fname $sitearchive/${fname}.new
fi

# Update the saved snapshot of the website.

mv $sitearchive/${fname}.new $sitearchive/$fname
chmod 755 $sitearchive/$fname
exit 0
```

Listing 7-20: The changetrack script

How It Works

Given a URL and a destination email address, this script grabs the web page content and compares it to the content of the site from the previous check. If the site has changed, the new web page is emailed to the specified recipient, with some simple rewrites to try to keep the graphics and HREFs working. These HTML rewrites starting at ❷ are worth examining.

The call to `curl` retrieves the source of the specified web page ❷, and then `sed` performs three different translations. First, `SRC="` is rewritten as `SRC="baseurl/` ❸ to ensure that any relative pathnames of the form `SRC="logo.gif"` are rewritten to work properly as full pathnames with the domain name. If the domain name of the site is *http://www.intuitive.com/*, the rewritten HTML would be `SRC="http://www.intuitive.com/logo.gif"`. Likewise, `HREF` attributes are rewritten ❹. Then, to ensure we haven't broken anything, the third translation pulls the `baseurl` back *out* of the HTML source in situations where it's been erroneously added ❺. For example, `HREF="http://www.intuitive.com/http://www.somewhereelse.com/link"` is clearly broken and must be fixed for the link to work.

Notice also that the recipient address is specified in the `echo` statement ❶ (`echo "To: $2"`) rather than as an argument to `sendmail`. This is a simple security trick: by having the address within the `sendmail` input stream (which `sendmail` knows to parse for recipients because of the `-t` flag), there's no worry about users playing games with addresses like `"joe;cat /etc/passwd|mail larry"`. This is a good technique to use whenever you invoke `sendmail` within shell scripts.

Running the Script

This script requires two parameters: the URL of the site being tracked (and you'll need to use a fully qualified URL that begins with `http://` for it to work properly) and the email address of the person (or comma-separated group of people) who should receive the updated web page, as appropriate. Or, if you'd prefer, just use `-` (a hyphen) as the email address, and the `diff` output will instead be displayed on screen.

The Results

The first time the script sees a web page, the page is automatically mailed to the specified user, as Listing 7-21 shows.

```
$ changetrack http://www.intuitive.com/ taylor@intuitive.com
Status: first visit to http://www.intuitive.com/. Copy archived for future
analysis.
```

Listing 7-21: Running the `changetrack` script for the first time

All subsequent checks on *http://www.intuitive.com/* will produce an email copy of the site only if the page has changed since the last invocation of the script. This change can be as simple as a single typo fix or as complex as a complete redesign. While this script can be used for tracking any website, sites that don't change frequently will probably work best: if the site is the BBC News home page, checking for changes is a waste of CPU cycles because this site is *constantly* updated.

If a site has not changed when the script is invoked the second time, the script has no output and sends no email to the specified recipient:

```
$ changetrack http://www.intuitive.com/ taylor@intuitive.com
$
```

Hacking the Script

An obvious deficiency in the current script is that it's hardcoded to look for *http://* links, which means it will reject any HTTP web pages served over HTTPS with SSL. Updating the script to work with both would require some fancier regular expressions, but is totally possible!

Another change to make the script more useful could be to have a granularity option that would allow users to specify that if only one line has changed, the script should not consider the website updated. You could implement this by piping the `diff` output to `wc -l` to count lines of output changed. (Keep in mind that `diff` generally produces *three* lines of output for each line changed.)

This script is also more useful when invoked from a `cron` job on a daily or weekly basis. We have similar scripts that run every night and send us updated web pages from various sites that we like to track.

A particularly interesting possibility is to modify this script to work off a data file of URLs and email addresses, rather than requiring those as input parameters. Drop that modified version of the script into a `cron` job, write a web-based front end to the utility (similar to shell scripts in Chapter 8), and you've just duplicated a function that some companies charge people money to use. No kidding.