9/Functions

Functions are the basic building blocks for Processing programs. They have appeared in every example we've presented. For instance, we've frequently used the `size()` function, the `line()` function, and the `fill()` function. This chapter shows how to write new functions to extend the capabilities of Processing beyond its built-in features.

The power of functions is modularity. Functions are independent software units that are used to build more complex programs—like LEGO bricks, where each type of brick serves a specific purpose, and making a complex model requires using the different parts together. As with functions, the true power of these bricks is the ability to build many different forms from the same set of elements. The same group of LEGOs that makes a spaceship can be reused to construct a truck, a skyscraper, and many other objects.

Functions are helpful if you want to draw a more complex shape like a tree over and over. The function to draw the tree shape would be made up of Processing's built-in fuctions, like `line()`, that create the form. After the code to draw the tree is written, you don't need to think about the details of tree drawing again—you can simply write `tree()` (or whatever you named the function) to draw the shape. Functions allow a complex sequence of statements to be abstracted, so you can focus on the higher-level goal (such as drawing a tree), and not the details of the implementation (the `line()` functions that define the tree

shape). Once a function is defined, the code inside the function need not be repeated again.

Function Basics

A computer runs a program one line at a time. When a function is run, the computer jumps to where the function is defined and runs the code there, then jumps back to where it left off.

Example 9-1: Roll the Dice

This behavior is illustrated with the `rollDice()` function written for this example. When a program starts, it runs the code in `setup()` and then stops. The program takes a detour and runs the code inside `rollDice()` each time it appears:

```
void setup() {
  println("Ready to roll!");
  rollDice(20);
  rollDice(20);
  rollDice(6);
  println("Finished.");
}

void rollDice(int numSides) {
  int d = 1 + int(random(numSides));
  println("Rolling... " + d);
}
```

The two lines of code in `rollDice()` select a random number between 1 and the number of sides on the dice, and prints that number to the Console. Because the numbers are random, you'll see different numbers each time the program is run:

```
Ready to roll!
Rolling... 20
Rolling... 11
Rolling... 1
Finished.
```

Each time the `rollDice()` function is run inside `setup()`, the code within the function runs from top to bottom, then the program continues on the next line within `setup()`.

The `random()` function returns a number from 0 up to (but not including) the number specified. So `random(6)` returns a number

between 0 and 5.99999. . . Because `random()` returns a `float` value, we also use `int()` to convert it to an integer. So `int(random(6))` will return 0, 1, 2, 3, 4, or 5. Then we add 1 so that the number returned is between 1 and 6 (like a die). Like many other cases in this book, counting from 0 makes it easier to use the results of `random()` with other calculations.

Example 9-2: Another Way to Roll

If an equivalent program were written without the `rollDice()` function, it might look like this:

```
void setup() {
  println("Ready to roll!");
  int d1 = 1 + int(random(20));
  println("Rolling... " + d1);
  int d2 = 1 + int(random(20));
  println("Rolling... " + d2);
  int d3 = 1 + int(random(6));
  println("Rolling... " + d3);
  println("Finished.");
}
```

The `rollDice()` function in Example 9-1 on page 122 makes the code easier to read and maintain. The program is clearer, because the name of the function clearly states its purpose. In this example, we see the `random()` function in `setup()`, but its use is not as obvious. The number of sides on the die is also clearer with a function: when the code says `rollDice(6)`, it's obvious that it's simulating the roll of a six-sided die. Also, it's easier to maintain Example 9-1 on page 122, because information is not repeated. The phase `Rolling...` is repeated three times here. If you want to change that text to something else, you would need to update the program in three places, rather than making a single edit inside the `rollDice()` function. In addition, as you'll see in Example 9-5 on page 126, a function can also make a program much shorter (and therefore easier to maintain and read), which helps reduce the potential number of bugs.

Make a Function

In this section, we'll draw an owl to explain the steps involved in making a function.

Example 9-3: Draw the Owl

First, we'll draw the owl without using a function:

```
void setup() {
  size(480, 120);
}

void draw() {
  background(176, 204, 226);
  translate(110, 110);
  stroke(138, 138, 125);
  strokeWeight(70);
  line(0, -35, 0, -65); // Body
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35);  // Left eye dome
  ellipse(17.5, -65, 35, 35);   // Right eye dome
  arc(0, -65, 70, 70, 0, PI);   // Chin
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8);  // Left eye
  ellipse(14, -65, 8, 8);   // Right eye
  quad(0, -58, 4, -51, 0, -44, -4, -51);  // Beak
}
```

Notice that `translate()` is used to move the origin (0,0) to 110 pixels over and 110 pixels down. Then the owl is drawn relative to (0,0), with its coordinates sometimes positive and negative as it's centered around the new 0,0 point. See Figure 9-1.

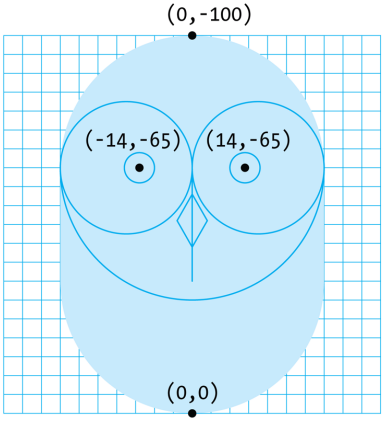
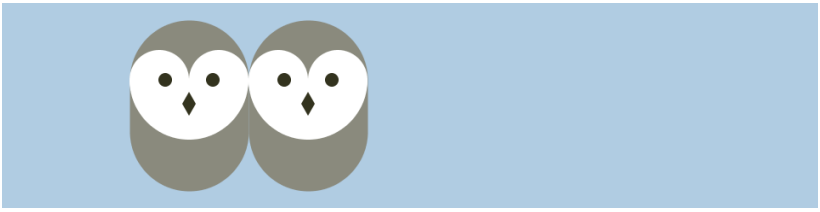


Figure 9-1. *The owl's coordinates*

Example 9-4: Two's Company

The code presented in Example 9-3 on page 124 is reasonable if there is only one owl, but when we draw a second, the length of the code is nearly doubled:



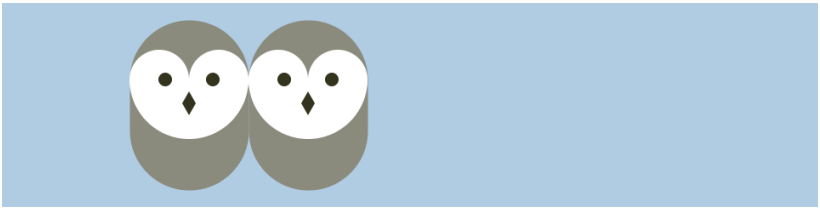
```
void setup() {
  size(480, 120);
}

void draw() {
  background(176, 204, 226);

  // Left owl
  translate(110, 110);
  stroke(138, 138, 125);
  strokeWeight(70);
  line(0, -35, 0, -65); // Body
  noStroke();
```

```
  fill(255);
  ellipse(-17.5, -65, 35, 35);  // Left eye dome
  ellipse(17.5, -65, 35, 35);   // Right eye dome
  arc(0, -65, 70, 70, 0, PI);   // Chin
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8);  // Left eye
  ellipse(14, -65, 8, 8);   // Right eye
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak

  // Right owl
  translate(70, 0);
  stroke(138, 138, 125);
  strokeWeight(70);
  line(0, -35, 0, -65); // Body
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35);  // Left eye dome
  ellipse(17.5, -65, 35, 35);   // Right eye dome
  arc(0, -65, 70, 70, 0, PI);   // Chin
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8);  // Left eye
  ellipse(14, -65, 8, 8);   // Right eye
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
}
```

The program grew from 21 lines to 34: the code to draw the first owl was cut and pasted into the program and a `translate()` was inserted to move it 70 pixels to the right. This is a tedious and inefficient way to draw a second owl, not to mention the headache of adding a third owl with this method. But duplicating the code is unnecessary, because this is the type of situation where a function can come to the rescue.

Example 9-5: An Owl Function

In this example, a function is introduced to draw two owls with the same code. If we make the code that draws the owl to the screen into a new function, the code need only appear once in the program:



```
void setup() {
  size(480, 120);
}

void draw() {
  background(176, 204, 226);
  owl(110, 110);
  owl(180, 110);
}

void owl(int x, int y) {
  pushMatrix();
  translate(x, y);
  stroke(138, 138, 125);
  strokeWeight(70);
  line(0, -35, 0, -65); // Body
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35); // Left eye dome
  ellipse(17.5, -65, 35, 35);  // Right eye dome
  arc(0, -65, 70, 70, 0, PI);  // Chin
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8); // Left eye
  ellipse(14, -65, 8, 8);  // Right eye
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
  popMatrix();
}
```

You can see from the illustrations that this example and Example 9-4 on page 125 have the same result, but this example is shorter, because the code to draw the owl appears only once, inside the aptly named `owl()` function. This code runs twice, because it's called twice inside `draw()`. The owl is drawn in two different locations because of the parameters passed into the function that set the *x* and *y* coordinates.

Parameters are an important part of functions, because they provide flexibility. We saw another example in the `rollDice()`

function; the single parameter named `numSides` made it possible to simulate a 6-sided die, a 20-sided die, or a die with any number of sides. This is just like many other Processing functions. For instance, the parameters to the `line()` function make it possible to draw a line from any pixel on screen to any other pixel. Without the parameters, the function would be able to draw a line only from one fixed point to another.

Each parameter has a data type (such as `int` or `float`), because each parameter is a variable that's created each time the function runs. When this example is run, the first time the `owl` function is called, the value of the `x` parameter is 110, and `y` is also 110. In the second use of the function, the value of `x` is 180 and `y` is again 110. Each value is passed into the function and then wherever the variable name appears within the function, it's replaced with the incoming value.

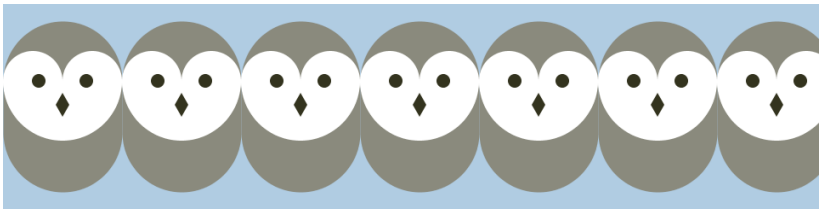
Make sure the values passed into a function match the data types of the parameters. For instance, if the following appeared inside the `setup()` for this example:

```
owl(110.5, 120.2);
```

This would create an error, because the data type for the `x` and `y` parameters is `int`, and the values 110.5 and 120.2 are `float` values.

Example 9-6: Increasing the Surplus Population

Now that we have a basic function to draw the owl at any location, we can draw many owls efficiently by placing the function within a `for` loop and changing the first parameter each time through the loop:

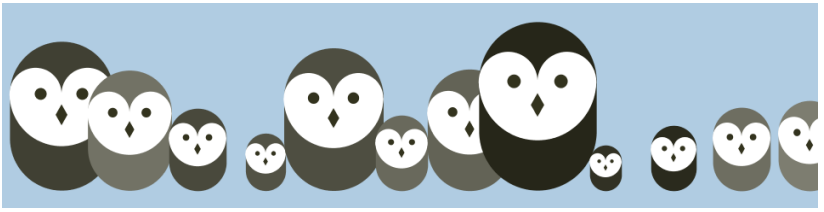


```
void setup() {
  size(480, 120);
}

void draw() {
  background(176, 204, 226);
  for (int x = 35; x < width + 70; x += 70) {
    owl(x, 110);
  }
}

// Insert owl() function from Example 9-5
```

It's possible to keep adding more and more parameters to the function to change different aspects of how the owl is drawn. Values could be passed in to change the owl's color, rotation, scale, or the diameter of its eyes.

Example 9-7: Owls of Different Sizes

In this example, we've added two parameters to change the gray value and size of each owl:



```
void setup() {
  size(480, 120);
}

void draw() {
  background(176, 204, 226);
  randomSeed(0);
  for (int i = 35; i < width + 40; i += 40) {
    int gray = int(random(0, 102));
    float scalar = random(0.25, 1.0);
    owl(i, 110, gray, scalar);
  }
}

void owl(int x, int y, int g, float s) {
```

```
  pushMatrix();
  translate(x, y);
  scale(s);  // Set the size
  stroke(138-g, 138-g, 125-g); // Set the color value
  strokeWeight(70);
  line(0, -35, 0, -65); // Body
  noStroke();
  fill(255);
  ellipse(-17.5, -65, 35, 35); // Left eye dome
  ellipse(17.5, -65, 35, 35);  // Right eye dome
  arc(0, -65, 70, 70, 0, PI);  // Chin
  fill(51, 51, 30);
  ellipse(-14, -65, 8, 8);  // Left eye
  ellipse(14, -65, 8, 8);   // Right eye
  quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
  popMatrix();
}
```

Return Values

Functions can make a calculation and then return a value to the main program. We've already used functions of this type, including `random()` and `sin()`. Notice that when this type of function appears, the return value is usually assigned to a variable:

```
float r = random(1, 10);
```

In this case, `random()` returns a value between 1 and 10, which is then assigned to the `r` variable.

A function that returns a value is also frequently used as a parameter to another function. For instance:

```
point(random(width), random(height));
```

In this case, the values from `random()` aren't assigned to a variable—they are passed as parameters to `point()` and used to position the point within the window.

Example 9-8: Return a Value

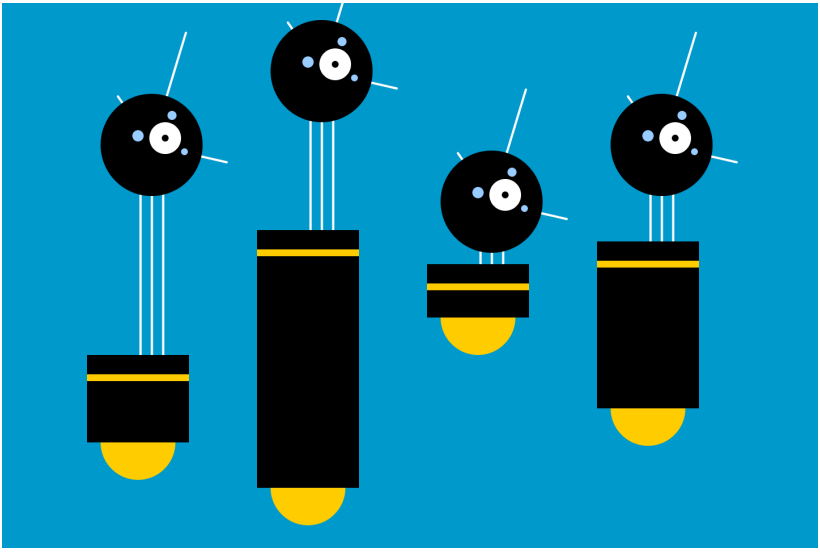
To make a function that returns a value, replace the keyword `void` with the data type that you want the function to return. In your function, specify the data to be passed back with the keyword `return`. For instance, this example includes a function

called `calculateMars()` that calculates the weight of a person or object on our neighboring planet:

```
void setup() {
  float yourWeight = 132;
  float marsWeight = calculateMars(yourWeight);
  println(marsWeight);
}

float calculateMars(float w) {
  float newWeight = w * 0.38;
  return newWeight;
}
```

Notice the data type `float` before the function name to show that it returns a floating-point value, and the last line of the block, which returns the variable `newWeight`. In the second line of `setup()`, that value is assigned to the variable `marsWeight`. (To see your own weight on Mars, change the value of the `yourWeight` variable to your weight.)

Robot 7: Functions



In contrast to Robot 2 (see "Robot 2: Variables" on page 47), this example uses a function to draw four robot variations within the same program. Because the `drawRobot()` function appears

four times within `draw()`, the code within the `drawRobot()` block is run four times, each time with a different set of parameters to change the position and height of the robot's body.

Notice how what were global variables in Robot 2 have now been isolated within the `drawRobot()` function. Because these variables apply only to drawing the robot, they belong inside the curly braces that define the `drawRobot()` function block. Because the value of the `radius` variable doesn't change, it need not be a parameter. Instead, it is defined at the beginning of `drawRobot()`:

```
void setup() {
  size(720, 480);
  strokeWeight(2);
  ellipseMode(RADIUS);
}

void draw() {
  background(0, 153, 204);
  drawRobot(120, 420, 110, 140);
  drawRobot(270, 460, 260, 95);
  drawRobot(420, 310, 80, 10);
  drawRobot(570, 390, 180, 40);
}

void drawRobot(int x, int y, int bodyHeight, int neckHeight) {

  int radius = 45;
  int ny = y - bodyHeight - neckHeight - radius;  // Neck y

  // Neck
  stroke(255);
  line(x+2, y-bodyHeight, x+2, ny);
  line(x+12, y-bodyHeight, x+12, ny);
  line(x+22, y-bodyHeight, x+22, ny);

  // Antennae
  line(x+12, ny, x-18, ny-43);
  line(x+12, ny, x+42, ny-99);
  line(x+12, ny, x+78, ny+15);

  // Body
  noStroke();
  fill(255, 204, 0);
```

```
  ellipse(x, y-33, 33, 33);
  fill(0);
  rect(x-45, y-bodyHeight, 90, bodyHeight-33);
  fill(255, 204, 0);
  rect(x-45, y-bodyHeight+17, 90, 6);

  // Head
  fill(0);
  ellipse(x+12, ny, radius, radius);
  fill(255);
  ellipse(x+24, ny-6, 14, 14);
  fill(0);
  ellipse(x+24, ny-6, 3, 3);
  fill(153, 204, 255);
  ellipse(x, ny-8, 5, 5);
  ellipse(x+30, ny-26, 4, 4);
  ellipse(x+41, ny+6, 3, 3);
}
```