Structure 3: Functions

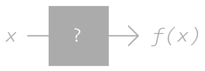
This unit introduces basic concepts and syntax for writing functions.

Syntax introduced:
`void, return`

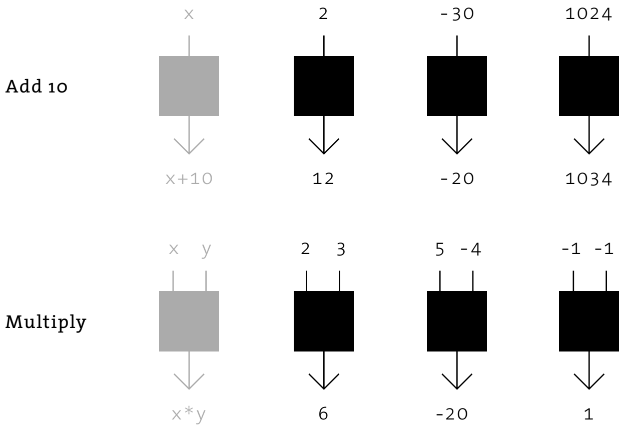
A function is a self-contained programming module. You've been using the functions included with Processing such as `size()`, `line()`, `stroke()`, and `translate()` to write your programs, but it's also possible to write your own functions that make a program modular. Functions make redundant code more concise by extracting the common elements and making them into code blocks that can be run many times within the program. This makes the code easier to read and update and reduces the chance of errors.

Functions often have parameters to define their actions. For example, the `line()` function has four parameters that define the position of the two points. Changing the numbers used as parameters changes the position of the line. Functions can operate differently depending on the number of parameters used. For example, a single parameter to the `fill()` function defines a gray value, two parameters define a gray value with transparency, and three parameters define an RGB color.

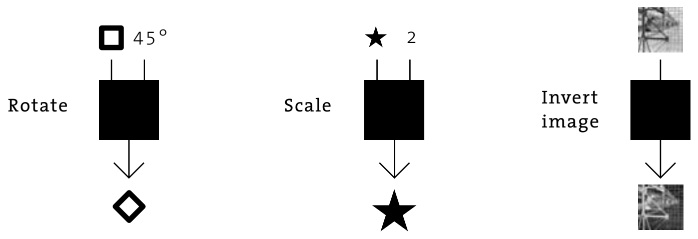
A function can be imagined as a box with mechanisms inside that act on data. There is typically an input into the box and code inside that utilizes the input to produce an output:



For example, a function can be written to add 10 to any number or to multiply two numbers:



The previous function examples are simple, but the concept can be extended to other processes that may be less obvious:



The mathematics used inside functions can be daunting, but the beauty of using functions is that it's not necessary to understand how they work. It's usually enough to know how to use them—to know what the inputs are and how they affect the output. This technique of ignoring the details of a process is called abstraction. It helps place the focus on the overall design of the program rather than the details.

Abstraction

In the terminology of software, the word *abstraction* has a different meaning from how it's used to refer to drawings and paintings. It refers to hiding details in order to focus on the result. The interface of the wheel and pedals in a car allows the driver to ignore details of the car's operation such as firing pistons and the flow of gasoline. The only understanding required by the person driving is that the steering wheel turns the vehicle left and right, the accelerator speeds it up, and the brake slows it down. Ignoring the minute details of the engine allows the driver to maintain focus on the task at hand. The mind need not be cluttered with thoughts about the details of execution.

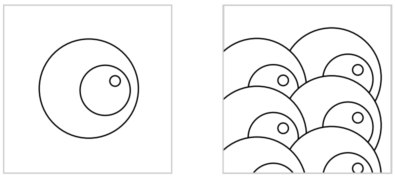
The idea of abstraction can also be discussed in relation to the human body. For example, we can control our breathing, but we usually breathe involuntarily, without conscious thought. Imagine if we had to directly control every aspect of our body. Having to continually control the beating of the heart, the release of chemicals, and the firing of neurons would make reading books and writing software impossible. The brain abstracts the basic functions of maintaining the body so our conscious minds can consider other aspects of life.

The idea of abstraction is essential to writing software. In Processing, drawing functions such as `line()`, `ellipse()`, and `fill()` obscure the complexity of their actions so that the author can focus on results rather than implementation. If you want to draw a line, you probably want to think only about its position, thickness, and color, and you don't want to think about the many lines of code that run behind the scenes to convert the line into a sequence of pixels.

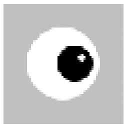
Creating functions

Before explaining in detail how to write your own functions, we'll first look at an example of why you might want to do so. The following examples show how to make a program shorter and more modular by adding a function. This makes the code easier to read, modify, and expand.

It's common to draw the same shape to the screen many times. We've created the shape you see below on the left, and now we want to draw it to the screen in the pattern on the right:



We start by drawing it once, to make sure our code is working.



21-01

```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

void draw() {
  fill(255);
  ellipse(50, 50, 60, 60);     // White circle
  fill(0);
  ellipse(50+10, 50, 30, 30);  // Black circle
  fill(255);
  ellipse(50+16, 45, 6, 6);    // Small, white circle
}
```

The previous program presents a sensible way to draw the shape once, but when another shape is added, we see a trend that continues for each additional shape. Adding a second shape inside `draw()` doubles the amount of code. Because it takes 6 lines to draw each shape, we now have 12 lines. Drawing our desired pattern that uses 6 shapes will require 36 lines of code. Imagine if we wanted to draw 30 eyes—the code inside `draw()` would bloat to 180 lines.

```
void setup() {                                          21-02
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

void draw() {
  // Right shape
  fill(255);
  ellipse(65, 44, 60, 60);
  fill(0);
  ellipse(75, 44, 30, 30);
  fill(255);
  ellipse(81, 39, 6, 6);
  // Left shape
  fill(255);
  ellipse(20, 50, 60, 60);
  fill(0);
  ellipse(30, 50, 30, 30);
  fill(255);
  ellipse(36, 45, 6, 6);
}
```

Because the shapes are identical, a function can be written for drawing them. The function introduced in the next example has two inputs that set the x-coordinate and y-coordinate. The lines of code inside the function render the elements for one shape.

```
void setup() {                                          21-03
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

void draw() {
  eye(65, 44);
  eye(20, 50);
}

void eye(int x, int y) {
  fill(255);
  ellipse(x, y, 60, 60);
  fill(0);
```

21-03 cont.

```
        ellipse(x+10, y, 30, 30);
        fill(255);
        ellipse(x+16, y-5, 6, 6);
      }
```

The function is 8 lines of code, but it only has to be written once. The code in the function runs each time it is referenced in `draw()`. Using this strategy, it would be possible to draw 30 eyes with only 38 lines of code.

A closer look at the flow of this program reveals how functions work and affect the program flow. Each time the function is used within `draw()`, the 6 lines of code inside the function block are run. The normal flow of the program is diverted by the function call, the code inside the function is run, and then the program returns to read the next line in `draw()`. Because `noLoop()` is used inside `setup()`, the lines of code in `draw()` only run once.

Start with code in `setup()`

```
size(100, 100)
noStroke()
smooth()
noLoop()
```

Enter `draw()`, divert to the eye function

```
fill(255)
ellipse(65, 44, 60, 60)
fill(0)
ellipse(75, 44, 30, 30)
fill(255)
ellipse(81, 39, 6, 6)
```

Back to `draw()`, divert to the eye function a second time

```
fill(255)
ellipse(20, 50, 60, 60)
fill(0)
ellipse(30, 50, 30, 30)
fill(255)
ellipse(36, 45, 6, 6)
```

Program ends

Now that the function is working, it can be used each time we want to draw that shape. If we want to use the shape in another program, we can copy and paste the function. We no longer need to think about how the shape is being drawn or what each line of code inside the function does. We only need to remember how to control its position with the two parameters.

21-04

```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

void draw() {
  eye(65, 44);
  eye(20, 50);
  eye(65, 74);
  eye(20, 80);
  eye(65, 104);
  eye(20, 110);
}

void eye(int x, int y) {
  fill(255);
  ellipse(x, y, 60, 60);
  fill(0);
  ellipse(x+10, y, 30, 30);
  fill(255);
  ellipse(x+16, y-5, 6, 6);
}
```

To write a function, start with a clear idea about what the function will do. Does it draw a specific shape? Calculate a number? Filter an image? After you know what the function will do, think about the parameters and the data type for each. Have a goal and break the goal into small steps.

In the following example, we first put together a program to explore some of the details of the function before writing it. Then, we start to build the function, adding one parameter at a time and testing the code at each step.

21-05

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  // Draw thick, light gray X
  stroke(160);
  strokeWeight(20);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
  // Draw medium, black X
  stroke(0);
  strokeWeight(10);
  line(30, 20, 90, 80);
  line(90, 20, 30, 80);
  // Draw thin, white X
  stroke(255);
  strokeWeight(2);
  line(20, 38, 80, 98);
  line(80, 38, 20, 98);
}
```

21-05 cont.

To write a function to draw the three X's in the previous example, first write a function to draw just one. We named the function `drawX()` to make its purpose clear. Inside, we have written code that draws a light gray X in the upper-left corner. Because this function has no parameters, it will always draw the same X each time its code is run. The keyword `void` appears before the function's name, which means the function does not return a value.

21-06

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX();
}

void drawX() {
  // Draw thick, light gray X
  stroke(160);
  strokeWeight(20);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}
```

To draw the X differently, add a parameter. In the next example the gray parameter variable has been added to the function to control the gray value of the X. The parameter variable must include its type and its name. When the function is called from within draw(), the value within the parentheses to the right of the function name is assigned to gray. In this example, the value 0 is assigned to gray, so the stroke is set to black.

21-07

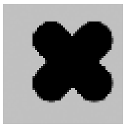
```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(0);  // Passes 0 to drawX(), runs drawX()
}

void drawX(int gray) {  // Declares and assigns gray
  stroke(gray);         // Uses gray to set the stroke
  strokeWeight(20);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}
```

A function can have more than one parameter. Each parameter for the function must be placed between the parentheses after the function name, each must state its data type, and the parameters must be separated by commas. In this example, the additional parameter weight is added to control the thickness of the line.

21-08

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(0, 30);  // Passes values to drawX(), runs drawX()
}

void drawX(int gray, int weight) {
  stroke(gray);
  strokeWeight(weight);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}
```

The next example extends `drawX()` to three additional parameters that control the position and size of the X drawn with the function.



21-09

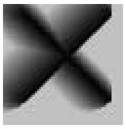
```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(0, 30, 40, 30, 36);
}

void drawX(int gray, int weight, int x, int y, int size) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+size, y+size);
  line(x+size, y, x, y+size);
}
```

By carefully building our function one step at a time, we have reached the original goal of writing a general function for drawing the three X's in code 21-05 (p. 186).

21-10

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(160, 20, 0, 5, 60);   // Draw thick, light gray X
  drawX(0, 10, 30, 20, 60);   // Draw medium, black X
  drawX(255, 2, 20, 38, 60);  // Draw thin, white X
}

void drawX(int gray, int weight, int x, int y, int size) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+size, y+size);
  line(x+size, y, x, y+size);
}
```

Now that we have the `drawX()` function, it's possible to write programs that would not be practical without it. For example, putting calls to `drawX()` inside a `for` structure allows for many repetitions. Each X drawn can be different from those previously drawn.



21-11

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  for (int i = 0; i < 20; i++) {
    drawX(200- i*10, (20-i)*2, i, i/2, 70);
  }
}

void drawX(int gray, int weight, int x, int y, int size) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+size, y+size);
  line(x+size, y, x, y+size);
}
```

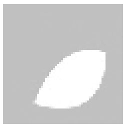
21-12

```
void setup() {
  size(100, 100);
  smooth();
  noLoop()
}

void draw() {
  for (int i = 0; i < 70; i++) {  // Draw 70 X shapes
    drawX(int(random(255)), int(random(30)),
          int(random(width)), int(random(height)), 100);
  }
}

void drawX(int gray, int weight, int x, int y, int size) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+size, y+size);
  line(x+size, y, x, y+size);
}
```

In the next series of examples, a `leaf()` function is created from code 7-17 (p. 77) to draw a leaf shape, and a `vine()` function is created to arrange a group of leaves onto a line. These examples demonstrate how functions can run inside other functions. The `leaf()` function has four parameters that determine the position, size, and orientation:

float x	*X-coordinate*
float y	*Y-coordinate*
float size	*Width of the leaf in pixels*
int dir	*Direction, either 1 (left) or -1 (right)*

This simple program draws one leaf and shows how the parameters affect its attributes.



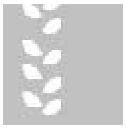
21-13

```
void setup() {
  size(100, 100);
  smooth();
  noStroke();
  noLoop();
}

void draw() {
  leaf(26, 83, 60, 1);
}

void leaf(int x, int y, int size, int dir) {
  pushMatrix();
  translate(x, y);  // Move to position
  scale(size);      // Scale to size
  beginShape();     // Draw the shape
  vertex(1.0*dir, -0.7);
  bezierVertex(1.0*dir, -0.7, 0.4*dir, -1.0, 0.0, 0.0);
  bezierVertex(0.0, 0.0, 1.0*dir, 0.4, 1.0*dir, -0.7);
  endShape();
  popMatrix();
}
```

The `vine()` function has parameters to set the position, the number of leaves, and the size of each leaf:

int x	*X-coordinate*
int numLeaves	*Total number of leaves on the vine*
float leafSize	*Width of the leaf in pixels*

This function determines the form of the vine by applying a few rules to the parameter values. The code inside `vine()` first draws a white vertical line, then determines the space between each leaf based on the height of the display window and the total number of leaves. The first leaf is set to draw to the right of the vine, and the `for` structure draws the number of leaves specified by the `numLeaves` parameter. The `x` parameter determines the position, and `leafSize` sets the size of each leaf. The y-coordinate of each leaf is slightly different each time the program is run because of the `random()` function.



21-14

```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  vine(33, 9, 16);
}

void vine(int x, int numLeaves, int leafSize ) {
  stroke(255);
  line(x, 0, x, height);
  noStroke();
  int gap = height / numLeaves;
  int direction = 1;
  for (int i = 0; i < numLeaves; i++) {
    int r = int(random(gap));
    leaf(x, gap*i + r, leafSize, direction);
    direction = -direction;
  }
}

// Copy and paste the leaf() function here
```

The `vine()` function was written in steps and was gradually refined to its present code. It could be extended with more parameters to control other aspects of the vine such as the color, or to draw on a curve instead of a straight line. In these examples, the vine function is called from `draw()` and the qualities of the vine are set by different parameters.

Shorter programs aren't the only benefit of using functions, but less code has advantages beyond a reduction in typing. Shorter programs lead to fewer errors—the more lines of code, the more chances for mistakes.

Imagine a novel written as a continuous paragraph without indentations or line breaks. Functions act as paragraphs that make your program easier to read. The practice of reducing complex processes into smaller, easier-to-comprehend units helps structure ideas. But it's not simply a matter of making lots of functions. Each function should be a unit of code that clearly expresses a single idea, calculation, or unit of form.

In code 21-02, six lines of code are needed to draw each shape. If we wanted to change one small detail—for example, the position of the small white circle in relation to the black circle—the corresponding line would need to be changed several times. If we were drawing nine shapes, nine lines of code would have to be changed. Once a group of code is put into a function, the program is easy to modify because that line of code need only be changed once.

Functions can make programs easier to write because they encourage reusing code. A custom function can be reused in another program. As you write more programs, you'll build a collection of functions that are useful across much of your work. In fact, parts of Processing evolved from function collections that the authors used in their own work.

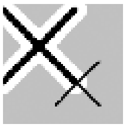
Function overloading

Multiple functions can have the same name, as long as they have different parameters. Creating different functions with the same name is called function overloading, and it's what allows Processing to have more than one version of functions like `fill()`, `image()`, and `text()`, each with different parameters. For example, the `fill()` function can have one, two, three, or four parameters. Each version of `fill()` sets the fill value for drawing shapes, but the number of parameters determines whether the fill value is gray, color, or includes transparency.

A program can also have two functions with the same number of parameters, but only if the data type for one of the parameters is different. For example, there are three versions of the `fill()` function with one parameter. The first uses a parameter of type `int` to set a gray value, the second uses a parameter of type `float` to set a gray value, and the third uses a parameter of type `color` to set a color value. The Processing language would be frustrating if a separate function name were used for each kind of fill.

This example uses three different `drawX()` functions, but all with the same name. The software knows which function to run by matching the number and type of the parameters.

21-15



```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

void draw() {
  drawX(255);  // Run first drawX()
  drawX(5.5);  // Run second drawX()
  drawX(0, 2, 44, 48, 36);  // Run third drawX()
}
```

21-15
cont.

```
// Draw an X with the gray value set by the parameter
void drawX(int gray) {
  stroke(gray);
  strokeWeight(20);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}

// Draw a black X with the thickness set by the parameter
void drawX(float weight) {
  stroke(0);
  strokeWeight(weight);
  line(0, 5, 60, 65);
  line(60, 5, 0, 65);
}

// Draws an X with the gray value, thickness,
// position, and size set by the parameters
void drawX(int gray, int weight, int x, int y, int s) {
  stroke(gray);
  strokeWeight(weight);
  line(x, y, x+s, y+s);
  line(x+s, y, x, y+s);
}
```

Calculating and returning values

In the examples shown so far, the output of a function has been shapes drawn to the screen. However, sometimes the preferred output is a number or other data. Data output from a function is called the return value. All functions are expected to return a value, such as an `int` or a `float`. If the function does not return a value, the special type `void` is used. The type of data returned by a function is found at the left of the function name.

The keyword `return` is used to exit a function and return to the location from which it was called. When a function outputs a value, `return` is used to specify what value should be returned. The `return` statement is typically the last line of a function because functions exit immediately after a `return`. We've already been using functions that return values. For example, `random()` returns a `float`, and the `color()` function returns a value of the `color` data type.

If a function returns a value, the function almost always appears to the right of an assignment operator or as a part of a larger expression. A function that does not return a value is often used as a complete statement. In the following example, notice how the value returned from the `random()` function is assigned to a variable, but the `ellipse()` function is not associated with a variable. If the `random()` function is not assigned to a variable, the value will be lost .

21-16
```
float d = random(0, 100);
ellipse(50, 50, d, d);
```

When using functions that return values, it's important to be aware of the data type that is returned by each function. For example, the `random()` function returns floating-point values. If the result of the `random()` function is assigned to an integer, an error will occur.

21-17
```
int d = random(0, 100); // ERROR! random() returns floats
ellipse(50, 50, d, d);
```

The data-type conversion functions (p. 105) are useful for converting data into the format needed within a program. The previous example can be modified with the `int()` conversion function to match the type of data returned from `random()` to the type of data the result is assigned to:

21-18
```
int d = int(random(0, 100)); // int() converts the float value
ellipse(50, 50, d, d);
```

Consult the reference for each function to learn what data type is returned. Functions are not limited to returning numbers: they can return a `PImage`, `String`, `boolean`, or any other data type.

To write your own functions that return values, replace `void` with the data type you want to return. Include the `return` keyword inside your function to set the data to output. The value of the expression following the `return` will be output from the function. The following examples make useful calculations and return values, so they can be used elsewhere in the program.

21-19
```
void setup() {
  size(100, 100);
  float f = average(12.0, 6.0);  // Assign 9.0 to f
  println(f);
}

float average(float num1, float num2) {
  float av = (num1 + num2) / 2.0;
  return av;
}
```

```
void setup() {                                                     21-20
  size(100, 100);
  float c = fahrenheitToCelsius(451.0); // Assign 232.77779 to c
  println(c);
}

float fahrenheitToCelsius(float t) {
  float f = (t-32.0) * (5.0/9.0);
  return f;
}
```

It's also important to note that you can't overload the return value of a function. Unlike functions that behave differently when given `float` or `int` values, it's not possible to have two functions with the same name that differ only in the type of data they return.

Exercises

1. *Write a function to draw a shape to the screen multiple times, each at a different position.*
2. *Extend the function created for exercise 1 by creating more parameters to control additional aspects of its form.*
3. *Write a function to use with a `for` structure to create a pattern evoking a liquid substance.*