# Structure 3: Functions

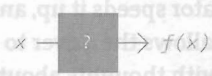
*This unit introduces basic concepts and syntax for writing functions.*

Syntax introduced:
`void`, `return`

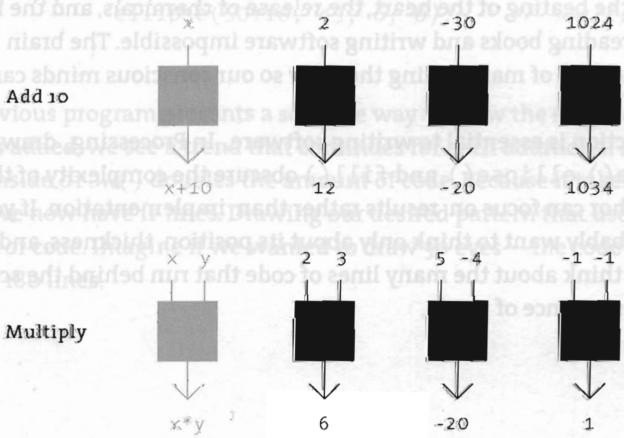
A function is a self-contained programming module. You've been using the functions included with Processing such as `size()`, `line()`, `stroke()`, and `translate()` to write your programs, but it's also possible to write your own functions that make a program modular. Functions make redundant code more 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.

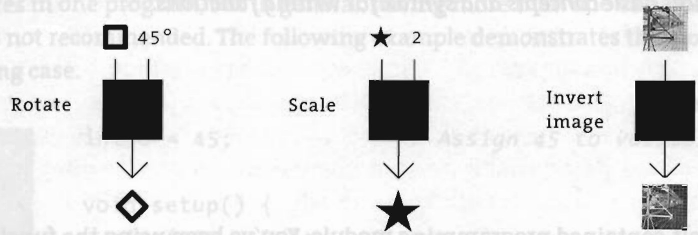
A function can be imagined as a box with mechanisms inside that act on data. There is typically an input into the box and code inside that utilizes the input to produce an output:




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.

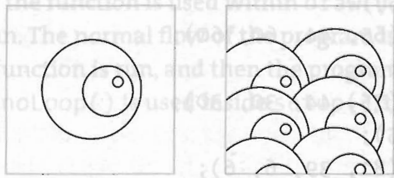
The idea of abstraction can also be discussed in relation to the human body. For example, we can control our breathing, but we usually breathe involuntarily, without conscious thought. Imagine if we had to directly control every aspect of our body. Having to continually control the beating of the heart, 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.

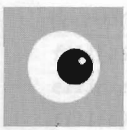
## Creating functions

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.

21-02

```
void setup() {
  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.

21-03

```
void setup() {
  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);
```

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

21-03 cont.

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

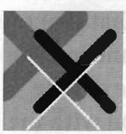
```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}

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

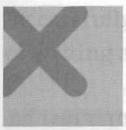
```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

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



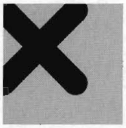
```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

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

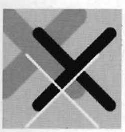
```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

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

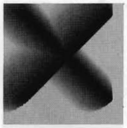


```
void setup() {
  size(100, 100);
  smooth();
  noLoop();
}

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

# Math 2: Curves

*This unit introduces drawing curves with mathematical equations.*

Syntax introduced:
`sq(), sqrt(), pow(), norm(), lerp(), map()`

Basic mathematical equations can be used to draw shapes to the screen and modify their attributes. These equations augment the drawing functions discussed in Shape 1 (p. 23) and Shape 2 (p. 69). They can control movement and the way elements respond to the cursor. This math is used to accelerate and decelerate shapes in motion and move objects along curved paths.

## Exponents, Roots

The `sq()` function is used to square a number and return the result. The result is always a positive number, because multiplying two negative numbers yields a positive result. For example, -1 * -1 = 1. This function has one parameter:

*sq(value)*

The value parameter can be any number. When `sq()` is used, the result can be assigned to a variable:

```
float a = sq(1);   // Assign 1 to a: Equivalent to 1 * 1       8-01
float b = sq(-5);  // Assign 25 to b: Equivalent to -5 * -5
float c = sq(9);   // Assign 81 to c: Equivalent to 9 * 9
```

The `sqrt()` function is used to calculate the square root of a number and return the result. It is the opposite of `sq()`. The square root of a number is always positive, even though there may be a valid negative root. The square root s of number a satisfies the equation s * s = a. This function has one parameter which must be a positive number:

*sqrt(value)*

As in the `sq()` function, the *value* parameter can be any number, and when the function is used the result can be assigned to a variable:

```
float a = sqrt(6561);  // Assign 81 to a       8-02
float b = sqrt(625);   // Assign 25 to b
float c = sqrt(1);     // Assign 1 to c
```

The pow() function calculates a number raised to an exponent. It has two parameters:

*pow(num, exponent)*

The *num* parameter is the number to multiply, and the *exponent* parameter is the number of times to make the calculation. The following example shows how it is used:

```
float a = pow(1, 3);    // Assign 1.0 to a: Equivalent to 1*1*1          8-03
float b = pow(3, 4);    // Assign 81.0 to b: Equivalent to 3*3*3*3
float c = pow(3, -2);   // Assign 0.11 to c: Equivalent to 1 / 3*3
float d = pow(-3, 3);   // Assign -27.0 to d: Equivalent to -3*-3*-3
```

Any number (except 0) raised to the zero power equals 1. Any number raised to the power of one equals itself.

```
float a = pow(8, 0);   // Assign 1 to a                                   8-04
float b = pow(3, 1);   // Assign 3 to b
float c = pow(4, 1);   // Assign 4 to c
```

## Normalizing, Mapping

Numbers are often converted to the range 0.0 to 1.0 for making calculations. This is called *normalizing* the values. When numbers between 0.0 and 1.0 are multiplied together, the result is never less than 0.0 or greater than 1.0. This allows any number to be multiplied by another or by itself many times without leaving this range. For example, multiplying the value 0.2 by itself 5 times (0.2 * 0.2 * 0.2 * 0.2 * 0.2) produces the result 0.00032. Because normalized numbers have a decimal point, all calculations should be made with the float data type.

To normalize a number, divide it by the maximum value that it represents. For example, to normalize a series of values between 0.0 and 255.0, divide each by 255.0:

| Initial value | Calculation | Normalized value |
|---|---|---|
| 0.0 | 0.0 / 255.0 | 0.0 |
| 102.0 | 102.0 / 255.0 | 0.4 |
| 255.0 | 255.0 / 255.0 | 1.0 |

This can also be accomplished via the norm() function. It has three parameters:

*norm(value, low, high)*

The number used as the *value* parameter is converted to a value between 0.0 and 1.0. The *low* and *high* parameters set the respective minimum and maximum values of the number's current range. If *value* is outside the range, the result may be less than 0 or greater than 1. The following example shows how to use the function to make the same calculations as the above table.

8-05

```
float x = norm(0.0, 0.0, 255.0);     // Assign 0.0 to x
float y = norm(102.0, 0.0, 255.0);   // Assign 0.4 to y
float z = norm(255.0, 0.0, 255.0);   // Assign 1.0 to z
```

After normalization, a number can be converted to another range through arithmetic. For example, to convert numbers between 0.0 and 1.0 in a range between 0.0 and 500.0, multiply by 500.0. To put numbers between 0.0 and 1.0 to numbers between 200.0 and 250.0, multiply by 50 then add 200. The following table presents a few sample conversions. The parentheses are used to improve readability:

| Initial range of x | Desired range of x | Conversion |
|---|---|---|
| 0.0 to 1.0 | 0.0 to 255.0 | x * 255.0 |
| 0.0 to 1.0 | -1.0 to 1.0 | (x * 2.0) - 1.0 |
| 0.0 to 1.0 | -20.0 to 60.0 | (x * 80.0) - 20.0 |

The `lerp()` function can be used to accomplish these calculations. The name "lerp" is a contraction for "linear interpolation." The function has three parameters:

*lerp(value1, value2, amt)*

The *value1* and *value2* parameters define the minimum and maximum values and the *amt* parameter defines the value to interpolate between the values. The *amt* parameter should always be a value between 0.0 and 1.0. The following example shows how to use `lerp()` to make the value conversions on the last line of the previous table.

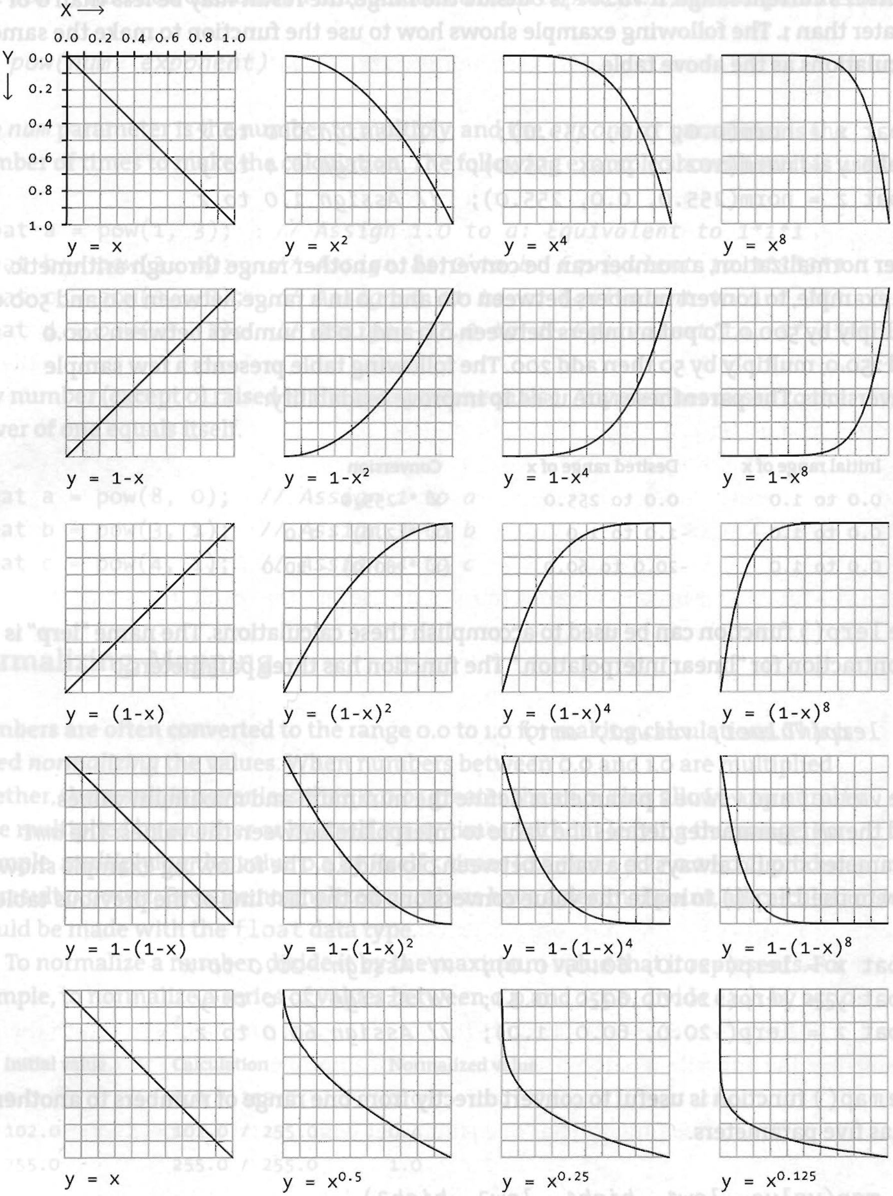
8-06

```
float x = lerp(-20.0, 60.0, 0.0);  // Assign -20.0 to x
float y = lerp(-20.0, 60.0, 0.5);  // Assign 20.0 to y
float z = lerp(-20.0, 60.0, 1.0);  // Assign 60.0 to z
```

The `map()` function is useful to convert directly from one range of numbers to another. It has five parameters.

*map(value, low1, high1, low2, high2)*

The *value* parameter is the number to re-map. Similar to the norm function, the *low1* and *low2* parameters are the minimum and maximum values of the number's current range. The *low2* and *high2* parameters are the minimum and maximum values for the new range. The next example shows how to use `map()` to convert values from the range 0 to 255 into the range -1 to 1. This is the same as first normalizing the value, then multiplying and adding to move it from the range 0 to 1 into the range -1 to 1.




Exponential equations

*Each of these curves shows the relationship between x and y determined by an equation. The linear equations in the left column are contrasted with exponential curves to the right. Codes 8-08 and 8-09 demonstrate how to translate these curves into code.*

8-07

```
float x = map(20.0, 0.0, 255.0, -1.0, 1.0);   // Assign -0.84 to x
float y = map(0.0, 0.0, 255.0, -1.0, 1.0);    // Assign -1.0 to y
float z = map(255.0, 0.0, 255.0, -1.0, 1.0);  // Assign 1.0 to z
```

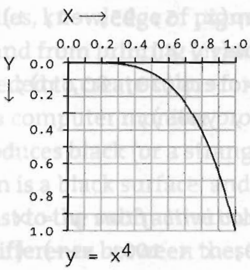
## Simple curves

Exponential functions are useful for creating simple curves. Normalized values are used with the pow() function to produce exponentially increasing or decreasing numbers that never exceed the value 1. These equations have the form:

$$y = x^n$$

where the value of x is between 0.0 and 1.0 and the value of n is any integer. In these equations, as the x value increases linearly the resulting y value increases exponentially. When continuously plotted, these numbers produce this diagram:

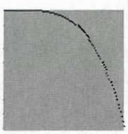
| X | Y |
|---|---|
| 0.0 | 0.0 |
| 0.2 | 0.0016 |
| 0.4 | 0.0256 |
| 0.6 | 0.1296 |
| 0.8 | 0.4096 |
| 1.0 | 1.0 |



$y = x^4$

The following example shows how to put this equation into code. It iterates over numbers from 0 to 100 and normalizes the values before making the curve calculation.

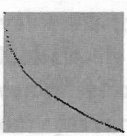
8-08



```
for (int x = 0; x < 100; x++) {
  float n = norm(x, 0.0, 100.0);  // Range 0.0 to 1.0
  float y = pow(n, 4);            // Calculate curve
  y *= 100;                       // Range 0.0 to 100.0
  point(x, y);
}
```

Other curves can be created by changing the parameters to pow() in line 3.

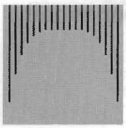
8-09



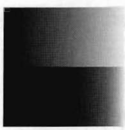
```
for (int x = 0; x < 100; x++) {
  float n = norm(x, 0.0, 100.0);  // Range 0.0 to 1.0
  float y = pow(n, 0.4);          // Calculate curve
  y *= 100;                       // Range 0.0 to 100.0
  point(x, y);
}
```

The following three examples demonstrate how the same curve is used to draw different shapes and patterns.

```
// Draw circles at points along the curve y = x^4                     8-10
noFill();
smooth();
for (int x = 0; x < 100; x += 5) {
  float n = norm(x, 0.0, 100.0); // Range 0.0 to 1.0
  float y = pow(n, 4);  // Calculate curve
  y *= 100;  // Scale y to range 0.0 to 100.0
  strokeWeight(n * 5);  // Increase thickness
  ellipse(x, y, 120, 120);
}
```



```
// Draw a line from the top of the display window to                  8-11
// points on a curve y = x^4 from x in range -1.0 to 1.0
for (int x = 5; x < 100; x += 5) {
  float n = map(x, 5, 95, -1, 1);
  float p = pow(n, 4);
  float ypos = lerp(20, 80, p);
  line(x, 0, x, ypos);
}
```



```
// Create a gradient from y = x and y = x^4                           8-12
for (int x = 0; x < 100; x++) {
  float n = norm(x, 0.0, 100.0);  // Range 0.0 to 1.0
  float val = n * 255.0;
  stroke(val);
  line(x, 0, x, 50);  // Draw top gradient
  float valSquare = pow(n, 4) * 255.0;
  stroke(valSquare);
  line(x, 50, x, 100);  // Draw bottom gradient
}
```

Exponential curves are used in this unit to generate form, but code 23-06 and 31-09 in subsequent units demonstrate their use to control motion and response.

Exercises

1. *Draw the curve $y = 1 - x^4$ to the display window.*
2. *Use the data from the curve $y = x^8$ to draw something unique.*
3. *Compose a range of gradients created from curves.*