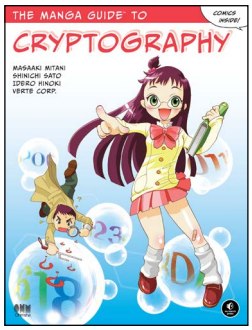
MORE FROM

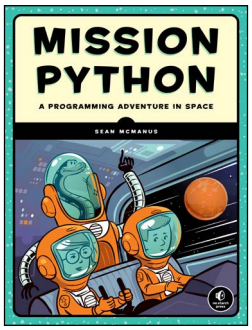
# NO STARCH PRESS

# COMING SOON FROM NO STARCH PRESS!




## THE MANGA GUIDE TO CRYPTOGRAPHY

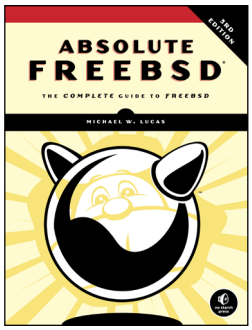
*by* MASAAKI MITANI, SHINICHI SATO, IDERO HINOKI, *and* VERTE CORP.
SUMMER 2018, 240 PP., $24.95
ISBN 978-1-59327-742-0




## MISSION PYTHON

### A Programming Adventure in Space

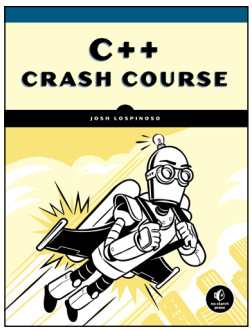
*by* SEAN MCMANUS
FALL 2018, 336 PP., $29.95
ISBN 978-1-59327-857-1
*full color*




## ABSOLUTE FREEBSD, 3RD EDITION

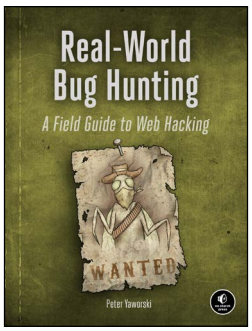
### The Complete Guide to FreeBSD

*by* MICHAEL W. LUCAS
SUMMER 2018, 832 PP., $59.95
ISBN 978-1-59327-892-2




## C++ CRASH COURSE

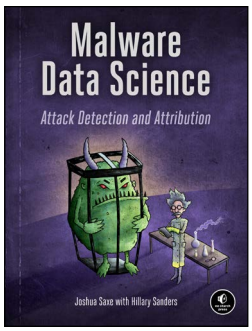
*by* JOSH LOSPINOSO
SPRING 2019, 524 PP., $49.95
ISBN 978-1-59327-888-5




## REAL-WORLD BUG HUNTING

### A Field Guide to Web Hacking

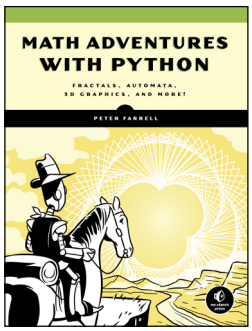
*by* PETER YAWORSKI
FALL 2018, 256 PP., $39.95
ISBN 978-1-59327-861-8




## MALWARE DATA SCIENCE

### Attack Detection and Attribution

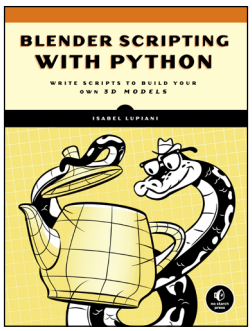
*by* JOSHUA SAXE *with* HILLARY SANDERS
SUMMER 2018, 400 PP., $49.95
ISBN 978-1-59327-859-5




## MATH ADVENTURES WITH PYTHON

### Fractals, Automata, 3D Graphics, and More!

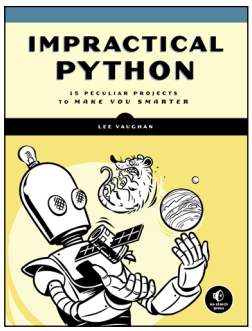
*by* PETER FARRELL
FALL 2018, 304 PP., $29.95
ISBN 978-1-59327-867-0
*full color*




## BLENDER SCRIPTING WITH PYTHON

### Write Scripts to Build Your Own 3D Models

*by* ISABEL LUPIANI
FALL 2018, 440 PP., $34.95
ISBN 978-1-59327-872-4




## IMPRACTICAL PYTHON

### 15 Peculiar Projects to Make You Smarter

*by* LEE VAUGHAN
SUMMER 2018, 504 PP., $29.95
ISBN 978-1-59327-890-8

# READ SAMPLE CHAPTERS FROM THESE NO STARCH BOOKS!

**20 EASY RASPBERRY PI PROJECTS**

RUI SANTOS AND SARA SANTOS

•

**THE LEGO ARCHITECTURE IDEA BOOK**

ALICE FINCH

•

**THE RUST PROGRAMMING LANGUAGE**

STEVE KLABNIK AND CAROL NICHOLS,
WITH CONTRIBUTIONS FROM THE RUST COMMUNITY

•

**CODING WITH MINECRAFT**

AL SWEIGART

•

**CRACKING CODES WITH PYTHON**

AL SWEIGART

•

**PRACTICAL SQL**

ANTHONY DEBARROS

•

**SERIOUS CRYPTOGRAPHY**

JEAN-PHILIPPE AUMASSON

•

**THE ARDUINO INVENTOR'S GUIDE**

BRIAN HUANG AND DEREK RUNBERG

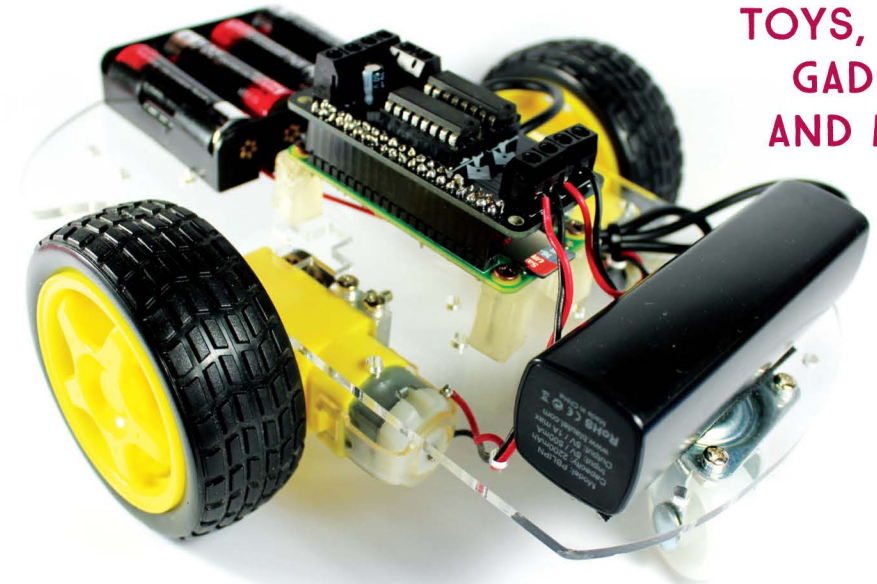
•

**LINUX BASICS FOR HACKERS**

OCCUPYTHEWEB

# 20 EASY RASPBERRY PI PROJECTS

TOYS, TOOLS, GADGETS, AND MORE!



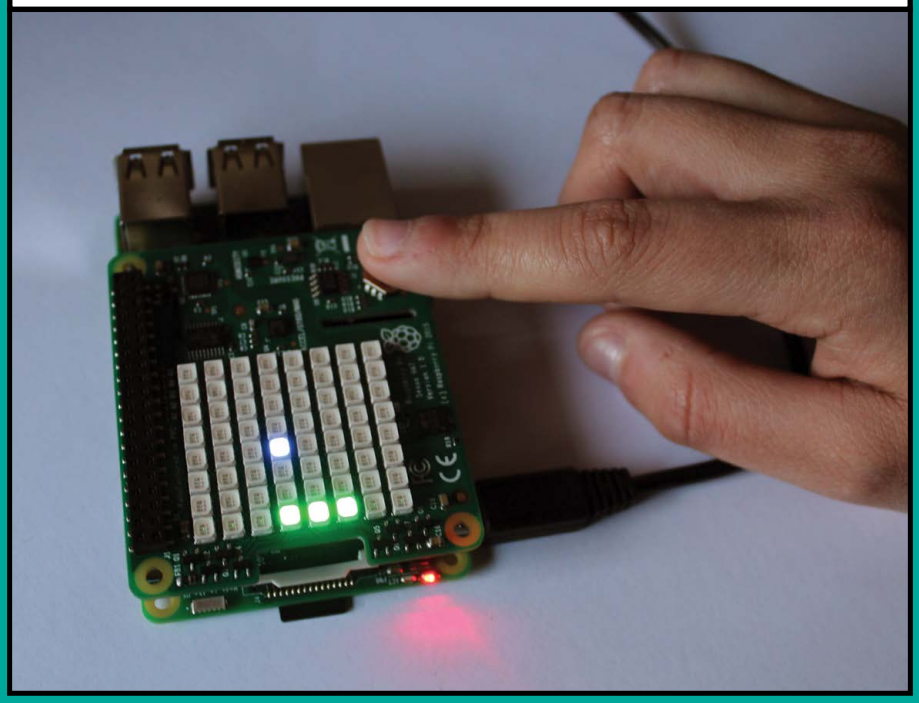
RUI SANTOS AND SARA SANTOS

no starch press

# 8
# PONG WITH A SENSE HAT

HERE YOU'LL BUILD YOUR OWN LED PONG GAME USING THE SENSE HAT. THE SENSE HAT IS AN ADD-ON BOARD FOR YOUR PI THAT GIVES IT A LOT MORE FUNCTIONALITY THROUGH EXTRA FEATURES LIKE AN LED MATRIX, JOYSTICK, AND SEVERAL SENSORS THAT GET INFORMATION FROM THE OUTSIDE WORLD.

COST: $$ TIME: 30 MINUTES



## PARTS REQUIRED

Raspberry Pi (versions with 40 GPIOs)
Sense HAT

You'll use the Sense HAT's LED matrix to display the game and the joystick to play. If you don't have the hardware, not to worry: you'll also learn how to use the Sense HAT emulator to create the same game without it.

## INTRODUCING PONG

One of the first video games ever created, Pong is an immensely popular 2D table-tennis (ping-pong) game that can be played in single- or double-player mode. You're going to create the single-player version, so it's more like playing squash: you bounce the ball against the walls with your bat and catch it with the bat when it comes back. If you miss the ball, you lose.

## INTRODUCING THE RASPBERRY PI SENSE HAT

The Raspberry Pi Sense HAT features an 8×8 RGB LED matrix, a five-button joystick, a gyroscope, an accelerometer, a magnetometer, a temperature sensor, a barometric sensor, and a humidity sensor in one package, shown in Figure 8-1.

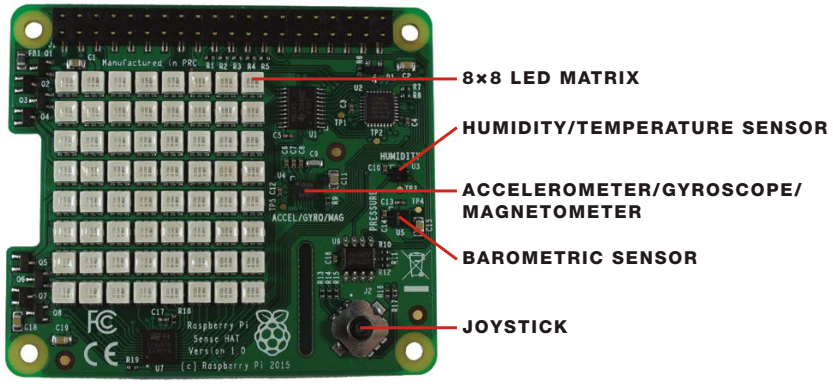



**FIGURE 8-1:** Raspberry Pi Sense HAT

### Mounting the Board

This project doesn't require much hardware assembly—you just need to mount the Sense HAT on the Pi, and the rest is done in code.

Attach the 40 GPIOs on the Sense HAT to the 40 GPIOs on your Raspberry Pi; the boards should line up perfectly. When you first successfully mount the Sense HAT on a powered Pi, the LED matrix displays an illuminated rainbow background as shown in Figure 8-2.

**NOTE**

*The Sense HAT is not compatible with Raspberry Pi 1 Model A and B, but you can build the project using the emulator if you have an incompatible board.*

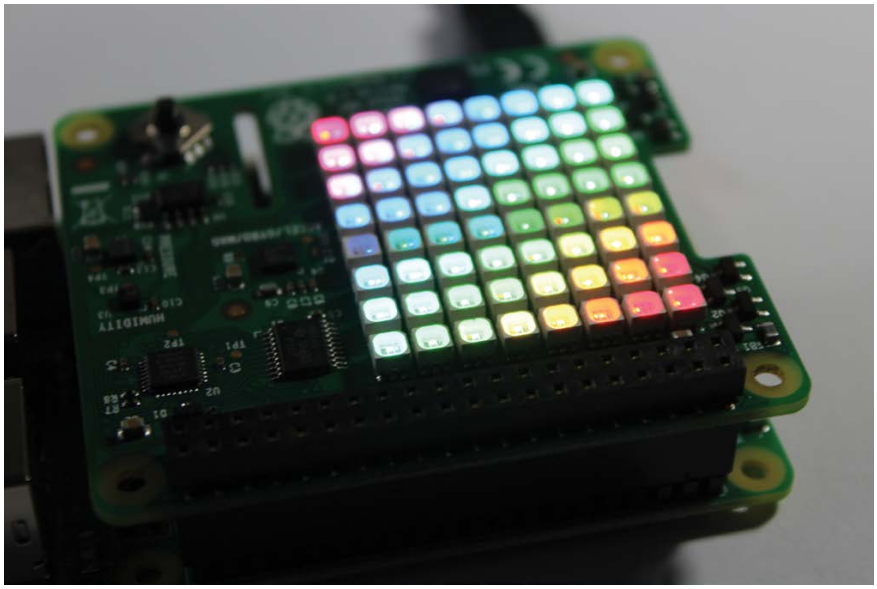


**FIGURE 8-2:**
Sense HAT welcome rainbow

## Using the Sense HAT Emulator

If you don't have a Sense HAT or a compatible board, or if you just want to test the script first, you can use the Sense HAT emulator to build the Pong game on your computer. The emulator is a virtual Sense HAT that you can interact with to test your scripts. To launch it from the Desktop main menu, go to **Programming ▸ Sense HAT Emulator**. This opens the emulator window, shown in Figure 8-3.

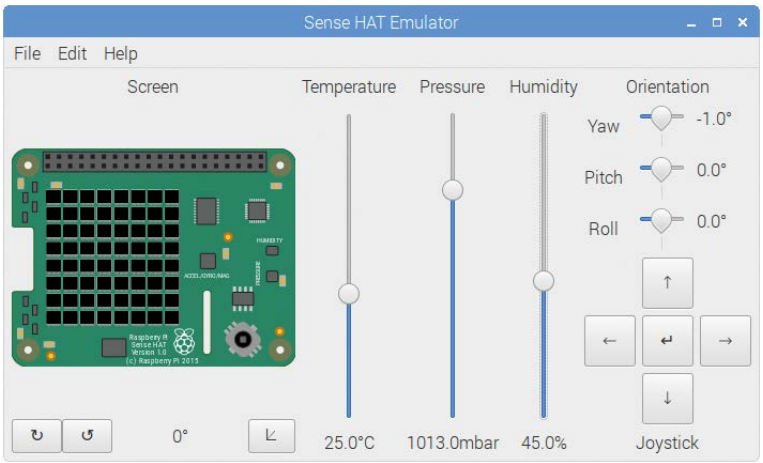



**FIGURE 8-3:**
Sense HAT Emulator window

The Sense HAT emulator comes with examples stored in **File ▸ Examples**; just select the example you want and then run the file to see the code in action in the emulator window.

## WORKING WITH SENSE HAT FUNCTIONS AND CONTROLS

Before you go right into building the game, it's important to understand how to control the LED matrix and read inputs from the joystick. Let's look at some examples that you'll use later in the Pong script.

### Controlling the LED Matrix

The Sense HAT LED matrix has 8 columns and 8 rows, containing a total of 64 RGB LEDs. You can display text and create images on the matrix by controlling each LED individually. You can also set the color of each LED.

#### Displaying Text

The code in Listing 8-1 displays the scrolling text "Hello World!" in blue on the dot matrix.

**LISTING 8-1:**
Display text on the Sense HAT LED matrix

```
❶ from sense_hat import SenseHat
  #uncomment the following line if you are using the emulator
❷ #from sense_emu import SenseHat
  sense = SenseHat()
❸ sense.show_message('Hello World!', text_colour = [0, 0, 255])
```

First import the `SenseHat` class ❶. If you're using the emulator, delete or comment out this line and uncomment the code at ❷.

The `show_message()` function ❸ accepts the message to display—a text string—as the first parameter, and then takes several options as further parameters:

**NOTE**
*The sense_hat library uses the British spelling "colour," so you must use "colour" throughout your code.*

- Use `text_colour = [r, g, b]` to set the RGB color of the text, replacing `r, g, b` with integers between `0` and `255` (as you did in Project 5).
- Use `scroll_speed = x`, where `x` is a float, to control the speed at which text moves across the display. The default scrolling speed is set to pause for 0.1 seconds each time the text shifts one pixel to the left.
- Use `back_colour = [r, g, b]` to set the background color, replacing `r, g, b` with integer values as with `text_colour`.

#### Controlling Specific LEDs

To control individual LEDs, you refer to each LED you want to light by its position in the matrix. For that, the Sense HAT uses an (x, y) coordinate system. For example, the LEDs in Figure 8-4 have the coordinates listed next to the diagram.

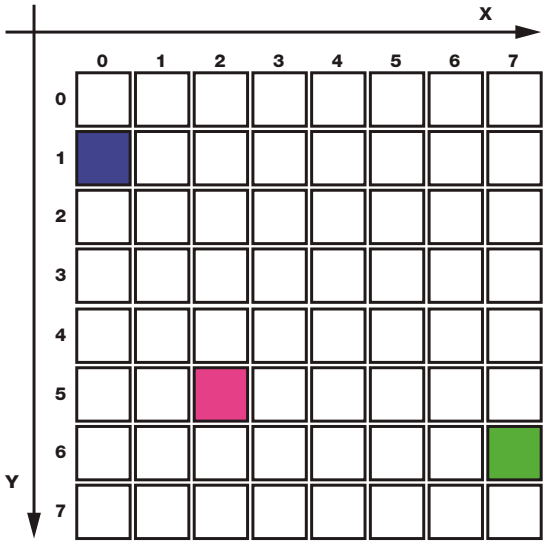



**FIGURE 8-4:**
Sense HAT coordinate system

To light up the LEDs in Figure 8-4 with their corresponding colors, you'd use the code in Listing 8-2.

```
from sense_hat import SenseHat
#uncomment the following line if you are using the emulator
#from sense_emu import SenseHat
sense = SenseHat()
#set blue pixel
sense.set_pixel(0, 1, 0, 0, 255)
#set green pixel
sense.set_pixel(7, 6, 0, 255, 0)
#set pink pixel
sense.set_pixel(2, 5, 255, 51, 153)
```

**LISTING 8-2:**
Using `set_pixel()` to light particular LEDs

The function `sense.set_pixel(x, y, r, g, b)` lights up a specific LED, in which *x* is the x-coordinate; *y* is the y-coordinate; and *r*, *g*, and *b* set the color.

## Displaying a Picture

Rather than controlling individual LEDs, you can use the function `sense.set_pixels()` to more quickly display an image. Instead of entering coordinates, you insert a list for all 64 LEDs that determines the color of each LED. Take a look at the code in Listing 8-3, which displays a sad face.

```
from sense_hat import SenseHat
#uncomment the following line if you are using the emulator
#from sense_emu import SenseHat
sense = SenseHat()

#red color
X = [255, 0, 0]
```

**LISTING 8-3:**
Displaying an image with `set_pixels()`

```
#no color
N = [0, 0, 0]

#sad face array
sad_face = [
N, N, X, X, X, X, N, N,
N, X, N, N, N, N, X, N,
X, N, X, N, N, X, N, X,
X, N, N, N, N, N, N, X,
X, N, N, X, X, N, N, X,
X, N, X, N, N, X, N, X,
N, X, N, N, N, N, X, N,
N, N, X, X, X, X, N, N
]

sense.set_pixels(sad_face)
```

**NOTE**

*The red Xs in the `sad_face` array won't appear red in your code. We're just highlighting them so it's easier to visualize how the LEDs will look.*

You create a variable to store the color of the lit LEDs (`X`), and a variable to store the color of the background (`N`)—you can set the background to any color or set it to `0` to keep it unlit. Then you need to create an array that sets each of the 64 LEDs either to `X` or to `N`. Figure 8-5 shows the end result of the code in Listing 8-3:

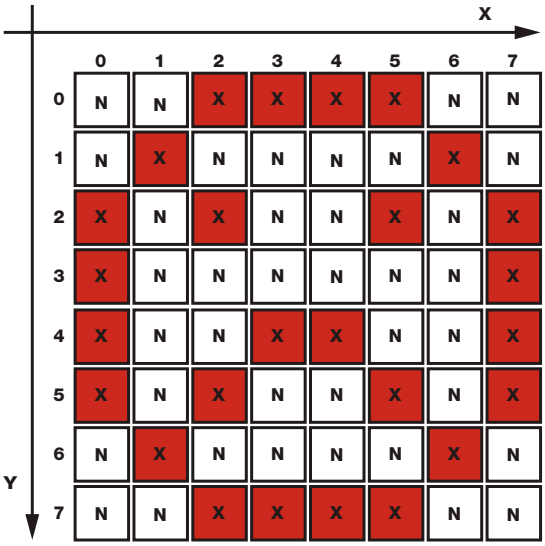



**FIGURE 8-5:** Displaying a sad face on the LED matrix

You can include as many colors as you want in your drawing; you just need to change the color parameters. We encourage you to practice working with the LED matrix by changing the colors and drawing your own images.

Now that you know how to control the LED matrix, let's look at how to program the joystick.

## Reading Data from the Joystick

The joystick that comes with the Sense HAT has five control options:

- Move up
- Move down
- Move right
- Move left
- Press

You need to tell your program what each control option should make the Pi do. The script in Listing 8-4 sets the events associated with each joystick control, and displays a message on the computer screen saying which control was used:

```
from signal import pause

from sense_hat import SenseHat
#uncomment the following line if you are using the emulator
#from sense_emu import SenseHat
sense = SenseHat()

❶ def move_up(event):
    print('joystick was moved up')

def move_down(event):
    print('joystick was moved down')

def move_right(event):
    print('joystick was moved right')

def move_left(event):
    print('joystick was moved left')

def move_middle(event):
    print('joystick was pressed')

❷ sense.stick.direction_up = move_up
sense.stick.direction_down = move_down
sense.stick.direction_right = move_right
sense.stick.direction_left = move_left
sense.stick.direction_middle = move_middle

pause()
```

**LISTING 8-4:**
Associating events with each joystick control

First, you need to tell your Pi what action to take when each joystick control is triggered. You do that by defining a series of functions to perform actions. For example, when the joystick is

moved up, you call the function `move_up()` ❶ to print the message `joystick was moved up`. The `event` argument tells the Pi that the joystick will be sending information to those functions. Then you use `sense.stick.direction_up = move_up` ❷ to associate the `move_up` function with the up movement of the joystick.

The other movement functions work in the same way.

## WRITING THE SCRIPT

Now that you know how to display text and drawings on the LED matrix and how to make something happen when the joystick is used, you're ready to start writing the script for your game.

Here's what the game aims to do:

- A bat that is 3 pixels long and 1 pixel wide should appear in column 0.
- Each time you move the joystick up or down, the bat should move correspondingly.
- The ball should start in a random position and move diagonally.
- When the ball hits something—walls, ceiling, or the bat—it should move diagonally in the opposite direction.
- If the ball hits column 0, it means you missed the ball, so you lose and the game is over.

### Entering the Script

Open **Python 3 (IDLE)** and go to **File ▸ New File** to create a new script. Then copy the code in Listing 8-5 to the new file and save the script as *pong_game.py* inside the *Displays* folder (remember that you can download all the scripts at *https://www.nostarch.com/RaspberryPiProject/*).

**LISTING 8-5:**
The Pong game code

```
#based on raspberrypi.org Sense HAT Pong example

#import necessary libraries
❶ from random import randint
from time import sleep

#use this line if you are using the Sense HAT board
from sense_hat import SenseHat
#uncomment the following line if you are using the emulator
#from sense_emu import SenseHat

#create an object called sense
❷ sense = SenseHat()
```

```
#set bat position, random ball position, and velocity
❸ y = 4
❹ ball_position = [int(randint(2,6)), int(randint(1,6))]
❺ ball_velocity = [1, 1]

#red color
X = [255, 0, 0]
#no color
N = [0, 0, 0]

#sad face array
sad_face = [
N, N, X, X, X, X, N, N,
N, X, N, N, N, N, X, N,
X, N, X, N, N, X, N, X,
X, N, N, X, N, N, N, X,
X, N, N, X, N, N, N, X,
X, N, X, N, N, X, N, X,
N, X, N, N, N, N, X, N,
N, N, X, X, X, X, N, N
]

#draw bat at y position
❻ def draw_bat():
    sense.set_pixel(0, y, 0, 255, 0)
    sense.set_pixel(0, y+1, 0, 255, 0)
    sense.set_pixel(0, y-1, 0, 255, 0)

#move bat up
❼ def move_up(event):
    global y
    if y > 1 and event.action=='pressed':
        y -= 1

#move bat down
def move_down(event):
    global y
    if y < 6 and event.action=='pressed':
        y += 1

#move ball to the next position
❽ def draw_ball():
    #ball displayed on current position
    sense.set_pixel(ball_position[0], ball_position[1], 75, 0, 255)
    #next ball position
    ball_position[0] += ball_velocity[0]
    ball_position[1] += ball_velocity[1]
    #if ball hits ceiling, calculate next position
    if ball_position[0] == 7:
        ball_velocity[0] = -ball_velocity[0]
    #if ball hits wall, calculate next position
    if ball_position[1] == 0 or ball_position[1] == 7:
        ball_velocity[1] = -ball_velocity[1]
```

```
        #if ball reaches 0 position, player loses and game quits
        if ball_position[0] == 0:
            sleep(0.25)
            sense.set_pixels(sad_face)
            quit()
        #if ball hits bat, calculate next ball position
        if ball_position[0] == 1 and y - 1 <= ball_position[1] <= y+1:
            ball_velocity[0] = -ball_velocity[0]

    #when joystick moves up or down, trigger corresponding function
❾   sense.stick.direction_up = move_up
    sense.stick.direction_down = move_down

    #run the game
❿   while True:
        sense.clear()
        draw_bat()
        draw_ball()
        sleep(0.25)
```

There's a lot going on in this code. Let's walk through it step by step.

### Importing Necessary Libraries

At ❶, you import the `randint()` function from the rand library to generate pseudorandom integers and the `sleep()` function from the time library to set delay times.

At ❷, you create an object called `sense` that will be used to refer to the Sense HAT throughout the code.

### Creating the Bat

The bat is a 3-pixel bar that moves up and down the leftmost column. At ❸, you define the bat's starting position at 4 pixels down from the top with `y = 4`. The complete bat is drawn in green within the `draw_bat()` function ❻, which adds one more pixel to the top of the starting position (`y - 1`) and to the bottom (`y + 1`) to make the bat 3 pixels long.

### Moving the Bat

The bat moves just on the y-axis, so its x-coordinate is always `0`, but its y-coordinate needs to change as the player moves the bat. In other words, the player can only move the bat up and down. The `move_up()` and `move_down()` functions, defined at ❼, control those movements. At ❾, you tell the Pi what action to take when the player moves the joystick up or down by calling `move_up()` and `move_down()`, respectively.

Take a closer look at the `move_up()` function (the `move_down()` function works in a similar way):

```
#move bat up
def move_up(event):
    global y
    if y > 1 and event.action=='pressed':
        y -= 1
```

The `move_up()` function accepts `event` as a parameter. Basically, the `event` parameter allows you to pass some information about the joystick to the function—such as the time the stick was used; the direction it was pushed; and if it was pressed, released, or held—so the Pi knows how to react.

When the player moves the joystick up, the function moves the y-coordinate of the bat up by subtracting `1` from the variable `y`. But first, the code checks that `y > 1`; otherwise, the bat may end up moving out of the matrix.

**HINT**

*Writing* `y -= 1` *in Python is equal to* `y = y - 1`.

## Declaring Variable Scope

Note that `y` is defined as a `global` variable. Not all variables in a program are accessible at all locations in the program, so there might be areas where it is invalid to call a certain variable. A variable's *scope* is the area of a program where it is accessible. In Python, there are two basic variable scopes: *local* and *global*.

A variable defined in the main code body is global, meaning it is accessible anywhere else in the code. A variable defined inside a function is local to that function, so what you do with the local variable inside the function has no effect on variables outside, even if they have the same name.

As you want `y` to be usable both inside the function where it is defined and throughout the code, it needs to be declared as `global`. Otherwise, when you move the joystick nothing will happen, because the `y` variable is just being changed inside the function and not in the main body of the code.

## Creating the Ball

To make a moving ball, you first need a starting position and a velocity. At ❹, you set the ball's starting position using a list. Lists are defined between square brackets, `[`*`0th element, 1st element, ..., nth element`*`]`, and each element is separated by a comma. The elements in the lists have *zero indexing*, meaning the index for the first element is 0, not 1. In this case, our 0th element is the x-position, and the 1st element is the y-position.

When you start the game, the ball is in a random position, generated by the `randint()` function. That random position can be between 1 and 6 for the y-axis and 2 and 6 for the x-axis. These numbers ensure that the ball doesn't start on the ceiling, walls, or next to the bat.

### Moving the Ball

Once you have a starting position for the ball, you need to give it a velocity ❺ to get it moving. You create a list for the ball's velocity in which the 0th element is the velocity for the x-coordinate and the 1st element is the velocity for the y-coordinate.

You need to add or subtract the velocity to or from the current ball position to make the ball move forward or backward, respectively. The `draw_ball()` function at ❽ is where you display and move the ball, which always moves in diagonals. If it goes forward it continues forward, and if it goes backward it continues backward, unless it hits the ceiling or the bat, in which case it goes in the opposite direction.

### Keeping the Game Running

Once everything is set up, you add a `while` loop to keep the game running ❿. The `while` loop starts by cleaning the display; then, it calls the function `draw_bat()` to draw the bat and `draw_ball()` to display the ball.

The `sleep()` function in the last line defines the time the ball takes to move to another position, so you can use this function to determine how fast the ball moves. If you increase the delay time, the game becomes slower and easier; if you decrease it, the game moves faster. We encourage you to experiment with different delay times.

## Running the Script

Congratulations! After a lot of programming, you have your reward: you can play Pong on your Sense HAT! Press **F5** or go to **Run ▸ Run Module** to run the script.

When you lose and the game ends, the LED matrix displays a sad face as shown in Figure 8-6.

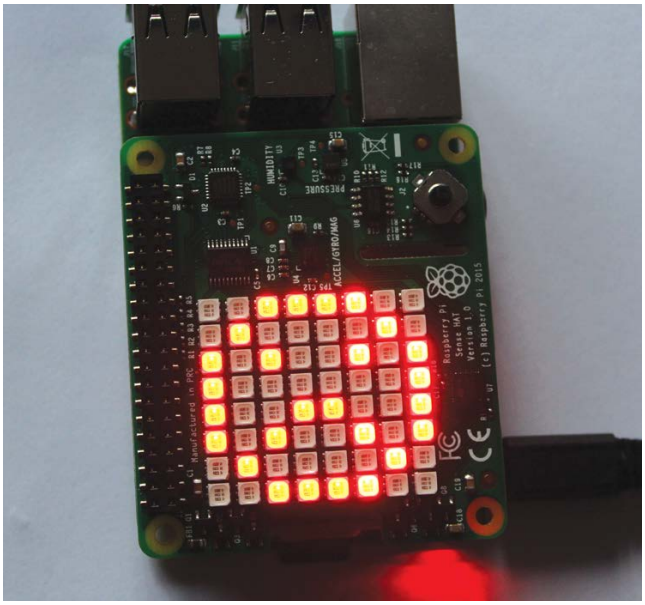


**FIGURE 8-6:**
LED matrix displaying a sad face when the game ends

## TAKING IT FURTHER

Here are some ideas to upgrade your game:

- Decrease the delay time as the game continues to increase the level of difficulty.
- Add a scoring system so that you earn a point every time the ball hits the bat, and display the score on the screen.
- Insert a condition that restarts the game when you press the joystick.

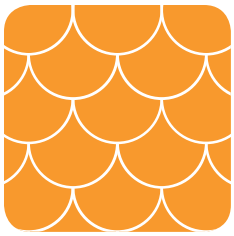
# THE LEGO® ARCHITECTURE IDEA BOOK

ALICE FINCH

1001 IDEAS FOR BRICKWORK, SIDING, WINDOWS, COLUMNS, ROOFING, AND MUCH, MUCH MORE!

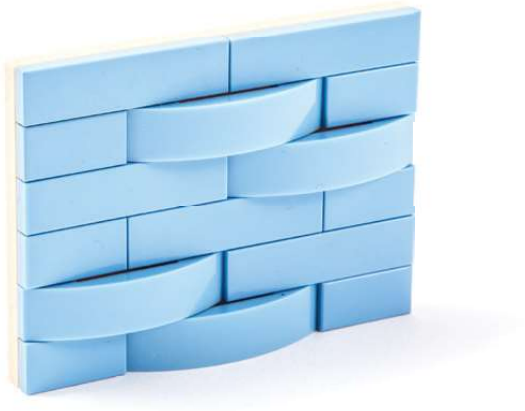
no starch press

# Wall Cladding



# Simple Siding

Tiles have a subtle groove at the bottom that you can use to emphasize gaps between the pieces. Mixing in curved slopes creates even more texture.



## Textured Siding

Wood grain tiles come in a variety of lengths and colors to choose from. You can use the same wood grain tiles to add wainscoting to a particular area of a wall. Third-party creators like Citizen Brick also create some interesting options.

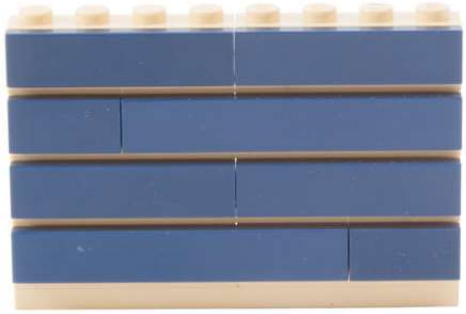
Gold bar pieces (#99563) also make for an interesting texture.

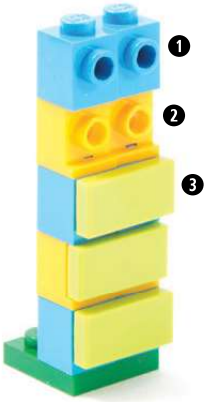
## Contrasting Siding

Attaching tiles to 1x2 jumper plates (#3794/#15573) and 2x2 jumper plates (#87580) creates a gap between each row of tiles. Use a contrasting color to emphasize the gap.

Using bricks with studs on the side instead of jumper plates creates a smaller gap.

1. 1x2 brick with 2 studs on the side #11211
2. 1x1 headlight brick #4070
3. 1x2 tile #3069

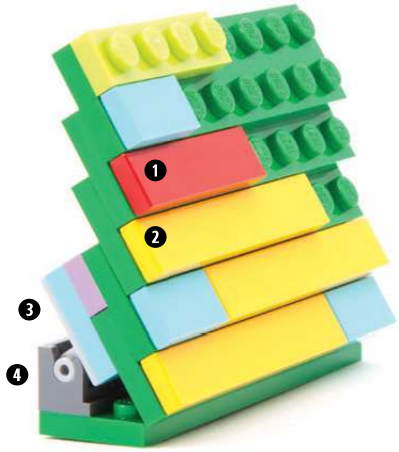





## Clapboard Siding

Tiles can also be attached to offset layers of 2x8 plates, which you can tilt at the base using a hinge to create beveled siding.

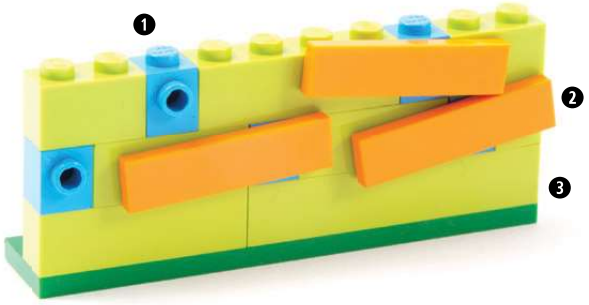
1. 1x4 tile #2431
2. 1x6 tile #6636
3. 2x2 hinge top #6134
4. 1x2 brick hinge base #3937



## Ramshackle Siding

Interspersing studded bricks with regular bricks gives you room to tilt the tiles, creating a worn down look.

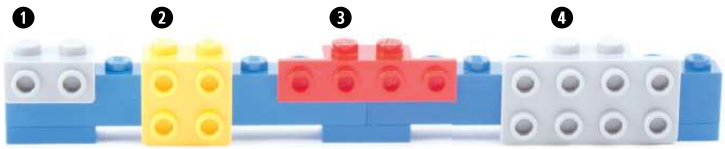
1. 1x1 with stud #87087
2. 1x4 tile #2431
3. Various 1x bricks



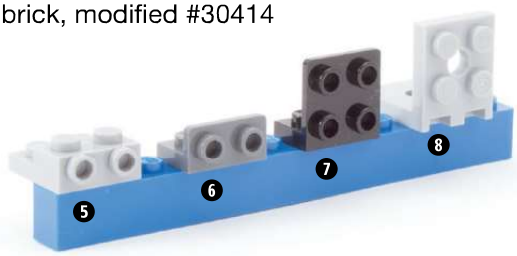

## Part Options

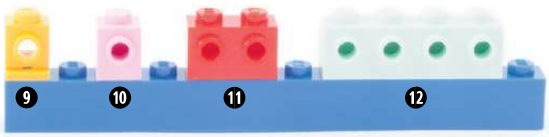
Many of these techniques use SNOT (studs not on top) and require building with sideways-facing studs. Here are a few ways to achieve that using bricks and brackets.

1. 1x2 - 2x1 bracket #99781
2. 1x2 - 2x2 bracket #44728
3. 1x2 -1x4 bracket #2436
4. 1x2 - 2x4 bracket #93274
5. 2x2x2/3 plate, modified #99206
6. 1x2 - 1x2 bracket, inverted #99780



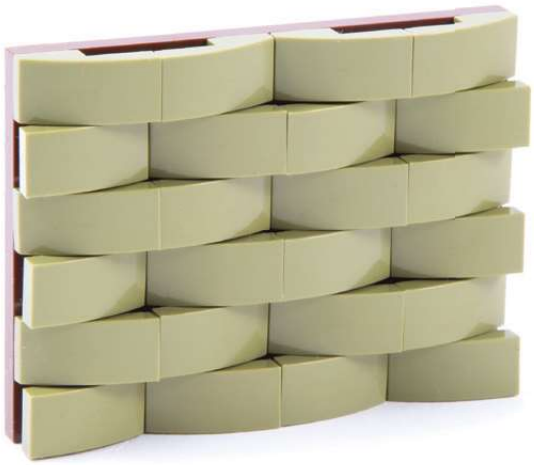

7. 1x2 - 2x2 bracket, inverted #99207
8. 2x2 - 2x2 bracket #3956
9. 1x1 headlight brick #4070
10. 1x2 brick with 1 stud #87087
11. 1x2 brick, modified#11211
12. 1x4 brick, modified #30414



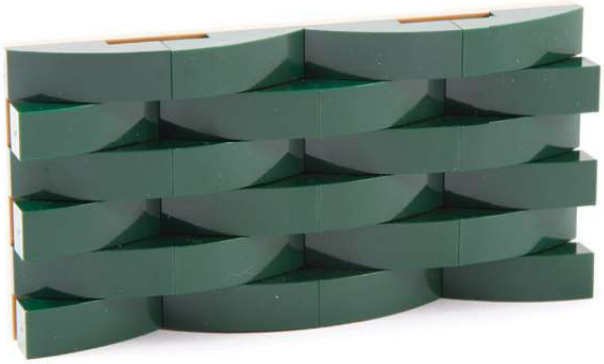
# Curved Slope Siding

Curved slopes create overlapping patterns that can be either brickwork or wood work, depending on their color.

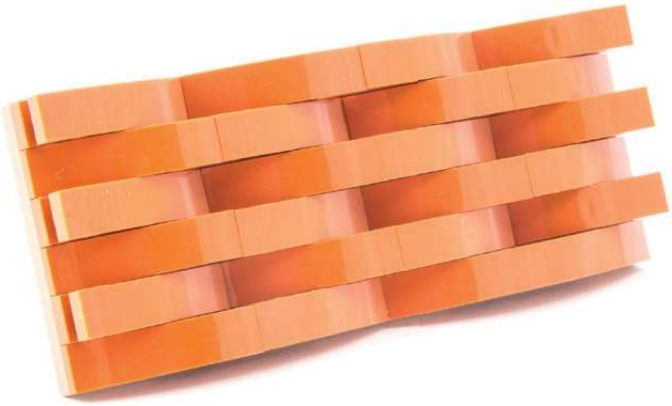


1x2 curved slope #11477

2x2 curved slope #15068



1x3 curved slope #50950

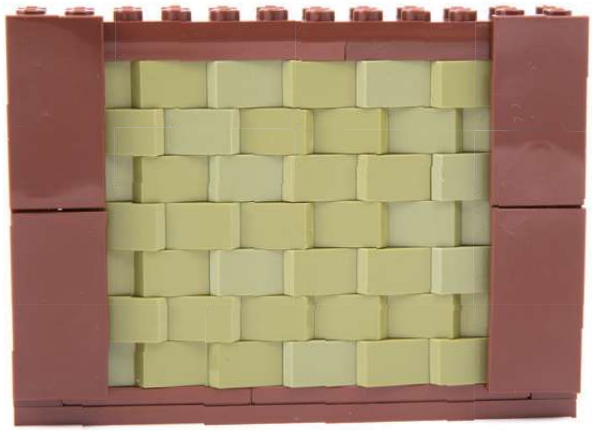


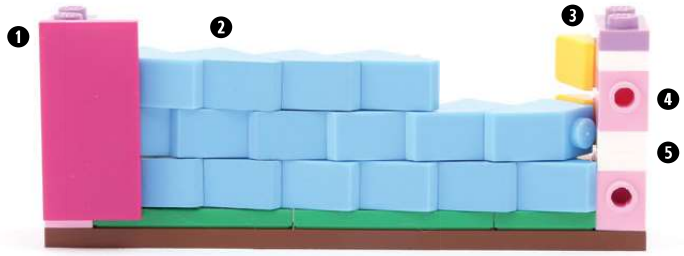
1x4 curved slope #61678

# Flat Slope Siding

Regular slopes have a flat lip that creates the look of overlapping siding.

For vertical siding, use a brick hinge (#3937/#3938 or #6134) to angle the slopes straight up.






For horizontal siding, alternate layers between supports that hold the layers firmly in place. It also looks good to frame the siding in a contrasting color.

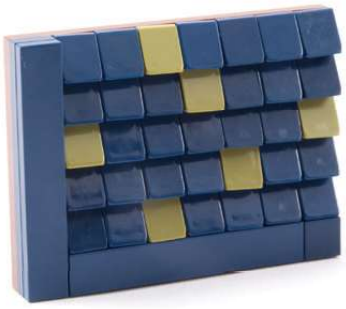
1. 2x4 tile #87079
2. 1x2 slope #3040
3. 1x1 cheese slope #54200
4. 1x1 brick with stud #87087
5. 1x1 plate

# Cheese Slope Siding

Use 1x1 cheese slopes to make siding or a shingled roof. The slopes can be set at a slight angle for an aged look thanks to the tolerances between parts.

Use complementary colors for a more random look or use contrasting colors to create a regular pattern.

Using 1x2 cheese slopes creates a smoother profile; the seams between the slopes can be staggered.



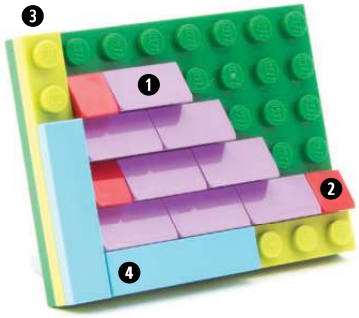
Using an occasional slope in another color gives it a weathered look, as if moss were growing on some of the shingles.

1x1 cheese slopes come in a huge variety of colors, with more added every year.

## How To

Frame the shingles with tiles to make it look different than the same part used on a roof.

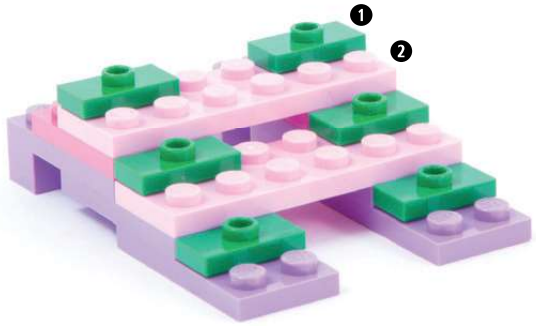
1. 1x2 double cheese slope #85984
2. 1x1 cheese slope #54200
3. 1x plates to raise tile edge
4. 1x4 tile border #2431



## Cheese Slope Roofs

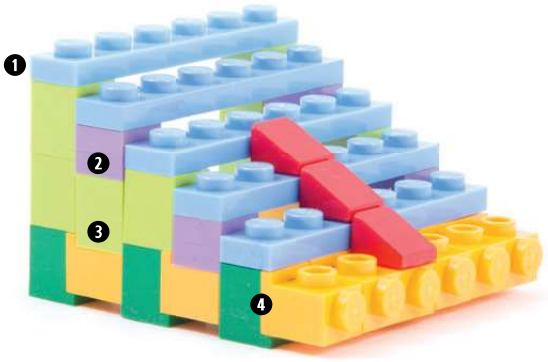
You can tweak this technique to cover not only flat surfaces but also sloping ones.

Attach 1x2 cheese slopes to jumper plates for an offset look.




1. 1x2 jumper plate #3794/#15573
2. 2x6 plate #3795

To create a smooth surface, use a mix of plates and bricks so that the 1x2 cheese slopes are level with the next row.




1. 1x6 plate #3666
2. 1x1 plate #3024
3. 1x1 brick #3005
4. 1x1 headlight brick #4070

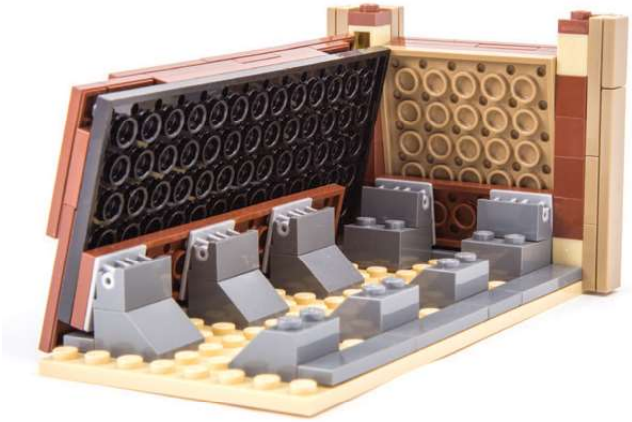
# Simple Shingle Siding

Arrange tiles vertically to create shingle patterns. Tiles can be partially or fully attached.

These tiles are only partially attached and they're vulnerable to coming off. The bottom layer can be straight or staggered.

These tiles are partially attached on alternating ends.

These tiles are fully attached to staggered layers of plates. As you can see, they're flush with the plates and securely attached, though the roof becomes thicker with each layer.



Tilt a tiled wall at an angle using brick hinges #3937 and #6134. Putting 2x2 slopes (#3039) under the hinges makes them makes them less likely to get pushed over.

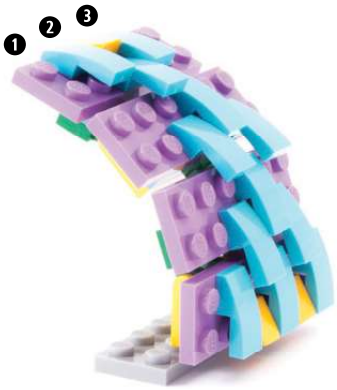
# Curved Shingle Siding

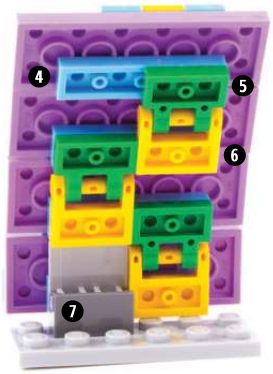
Make a curved shingle wall using rows of 1x2 curved slopes on 2x plates connected to plates with clips and handles.

The curved slopes are staggered to cover the gaps between the plates, and the curved wall is attached to the base with a brick hinge.

## How To

The use of 1x2 plates with 2 clips (#60470) and 1x2 plates with handle on side (#48336) make for a very flexible system for curving a wall at the desired angle without collapse.

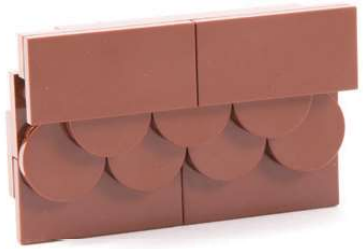



1. 2x plates
2. 1x2 curved slope #11477
3. 1x1 cheese slope #54200
4. 1x4 plate (for clearance)
5. 1x2 plate with 2 clips #60470
6. 1x2 plate with handle on side #48336
7. 1x2 hinge brick #3937 & #6134

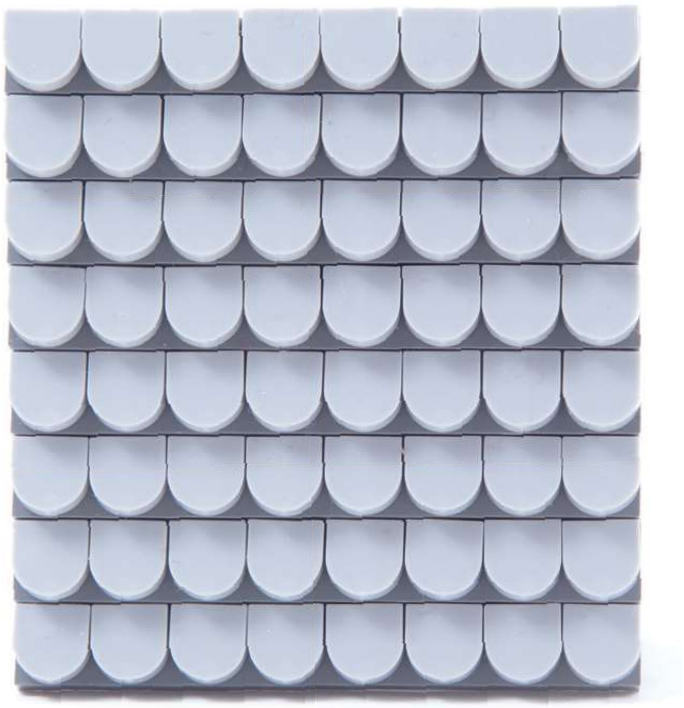
# Fishscale Shingle Siding

Round tiles and pentagonal tiles can be used for an entire wall or as a decorative shingle border or feature.



2x4 tiles #87079
2x2 round tiles #4150/14769

2x2 round tiles #4150/14769



1x1 half rounded tile #24246

2x3 pentagonal tile #22385

2x2 round tile #4150/14769
2x2 macaroni tile #27925

## Fishscale Shingle Roofs

You can use round tiles to cover sloped surfaces as well.

Layer round tiles (#4150) and round tiles with bottom stud holder (#14769) to create a stepped roof.

Use colorful tiles for a gingerbread house.

You can achieve a similar look using 2x2 plates (#4032), which look best when offset by one stud and with color variation.

3x2 modified plate with hole #3176

# Clip-on Shingle Siding

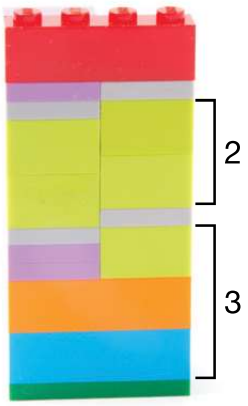
Tiles with clips create secure shingles that you can stagger to create rustic siding.

This shingle technique is 2 studs deep. You can cover the edges and corners by stacking 2x2x2 containers (#61780).

## How To

A 2x4 brick is used on the bottom so that each row of tiles sits at an angle. On the left, 2x3 tiles with 2 clips #30350 are clipped to 1x2 plate with handle on the side #48336.

You can also stagger the height of shingles using plates, as shown on the right. The first layer of gray plate with handle sits 3 bricks from the bottom. Successive layers of tiles are then 2 bricks apart. Adjacent plates can be staggered up or down one or two plates (lavender).



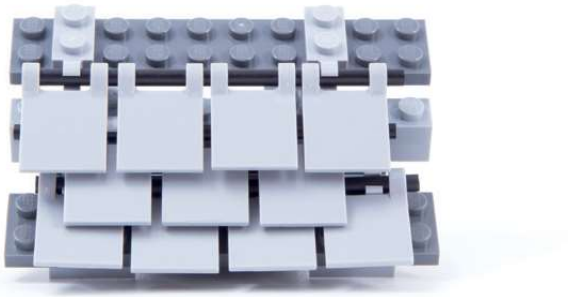
## Clip-on Shingle Roofs

To modify this technique for roofs or other sloped surfaces, you can layer pentagonal tiles on staggered plates to create a nice pattern.

2x2 trapezoid flag #44676

2x3 tile with 2 clips #30350

1x2 plate with handle on side #48336



2x2 square flag #2335

2x2 round sign #30261

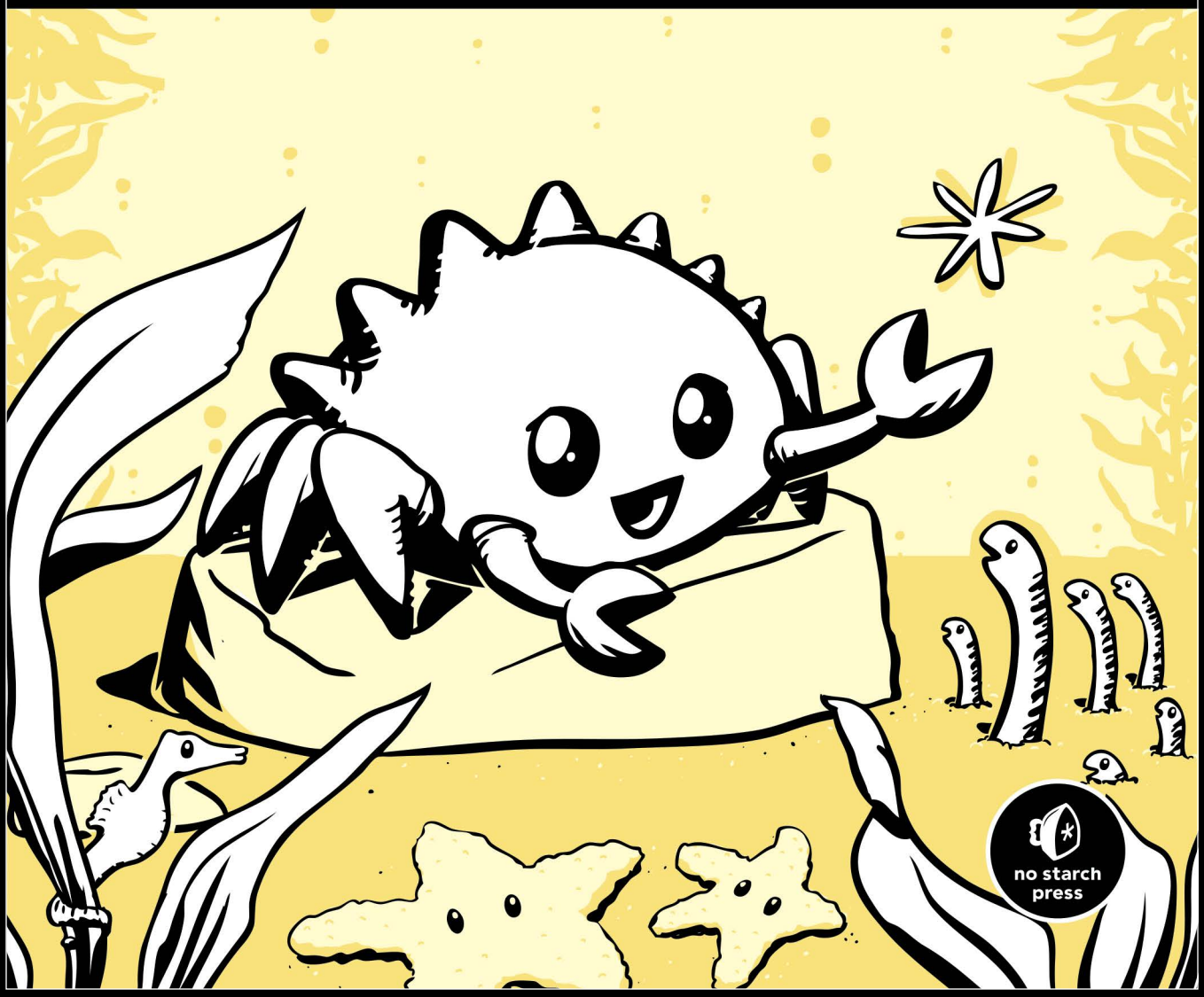
2x2 triangular sign #892

2x2 square sign #30258

Any of these tiles can attach to handles or to a 3 mm hose.

# THE RUST PROGRAMMING LANGUAGE

STEVE KLABNIK AND CAROL NICHOLS,
WITH CONTRIBUTIONS FROM THE RUST COMMUNITY



# 2

## PROGRAMMING A GUESSING GAME

Let's jump into Rust by working through a hands-on project together! This chapter introduces you to a few common Rust concepts by showing you how to use them in a real program. You'll learn about `let`, `match`, methods, associated functions, external crates, and more! The following chapters will explore these ideas in more detail. In this chapter, you'll practice the fundamentals.

We'll implement a classic beginner programming problem: a guessing game. Here's how it works: the program will generate a random integer between 1 and 100. It will then prompt the player to enter a guess. After a guess is entered, the program will indicate whether the guess is too low or too high. If the guess is correct, the game will print a congratulatory message and exit.

## Setting Up a New Project

To set up a new project, go to the *projects* directory that you created in Chapter 1 and make a new project using Cargo, like so:

```
$ cargo new guessing_game --bin
$ cd guessing_game
```

The first command, `cargo new`, takes the name of the project (`guessing_game`) as the first argument. The `--bin` flag tells Cargo to make a binary project, like the one in Chapter 1. The second command changes to the new project's directory.

Look at the generated *Cargo.toml* file:

*Cargo.toml*

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

If the author information that Cargo obtained from your environment is not correct, fix that in the file and save it again.

As you saw in Chapter 1, `cargo new` generates a "Hello, world!" program for you. Check out the *src/main.rs* file:

*src/main.rs*

```
fn main() {
    println!("Hello, world!");
}
```

Now let's compile this "Hello, world!" program and run it in the same step using the `cargo run` command:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev[unoptimized + debug info] target(s) in 1.50 sec
     Running `target/debug/guessing_game`
Hello, world!
```

The `run` command comes in handy when you need to rapidly iterate on a project, as we'll do in this game, quickly testing each iteration before moving on to the next one.

Reopen the *src/main.rs* file. You'll be writing all the code in this file.

## Processing a Guess

The first part of the guessing game program will ask for user input, process that input, and check that the input is in the expected form. To start, we'll allow the player to input a guess. Enter the code in Listing 2-1 into *src/main.rs*.

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

*Listing 2-1: Code that gets a guess from the user and prints it*

This code contains a lot of information, so let's go over it line by line. To obtain user input and then print the result as output, we need to bring the `io` (input/output) library into scope. The `io` library comes from the standard library (which is known as `std`):

```
use std::io;
```

By default, Rust brings only a few types into the scope of every program in the *prelude*. If a type you want to use isn't in the prelude, you have to bring that type into scope explicitly with a `use` statement. Using the `std::io` library provides you with a number of useful features, including the ability to accept user input.

As you saw in Chapter 1, the `main` function is the entry point into the program:

```
fn main() {
```

The `fn` syntax declares a new function, the parenthese, `()`, indicate there are no arguments, and the curly bracket, `{`, starts the body of the function.

As you also learned in Chapter 1, `println!` is a macro that prints a string to the screen:

```
println!("Guess the number!");

println!("Please input your guess.");
```

This code is printing a prompt stating what the game is and requesting input from the user.

## Storing Values with Variables

Next, we'll create a place to store the user input, like this:

```
let mut guess = String::new();
```

Now the program is getting interesting! There's a lot going on in this little line. Notice that this is a `let` statement, which is used to create a *variable*. Here's another example:

```
let foo = bar;
```

This line creates a new variable named `foo` and binds it to the value `bar`. In Rust, variables are immutable by default. The following example shows how to use `mut` before the variable name to make a variable mutable:

```
let foo = 5; // immutable
let mut bar = 5; // mutable
```

**NOTE** *The `//` syntax starts a comment that continues until the end of the line. Rust ignores everything in comments, which are discussed in more detail in Chapter 3.*

Now you know that `let mut guess` will introduce a mutable variable named `guess`. On the other side of the equal sign (`=`) is the value that `guess` is bound to, which is the result of calling `String::new`, a function that returns a new instance of a `String`. `String` is a string type provided by the standard library that is a growable, UTF-8 encoded bit of text.

The `::` syntax in the `::new` line indicates that `new` is an *associated function* of the `String` type. An associated function is implemented on a type, in this case `String`, rather than on a particular instance of a `String`. Some languages call this a *static method*.

This `new` function creates a new, empty string. You'll find a `new` function on many types, because it's a common name for a function that makes a new value of some kind.

To summarize, the `let mut guess = String::new();` line has created a mutable variable that is currently bound to a new, empty instance of a `String`. Whew!

Recall that we included the input/output functionality from the standard library with `use std::io;` on the first line of the program. Now we'll call an associated function, `stdin`, on `io`:

```
io::stdin().read_line(&mut guess)
    .expect("Failed to read line");
```

If we hadn't listed the `use std::io` line at the beginning of the program, we could have written this function call as `std::io::stdin`. The `stdin` function returns an instance of `std::io::Stdin`, which is a type that represents a handle to the standard input for your terminal.

The next part of the code, `.read_line(&mut guess)`, calls the `read_line` method on the standard input handle to get input from the user. We're also passing one argument to `read_line`: `&mut guess`.

The job of `read_line` is to take whatever the user types into standard input and place that into a string, so it takes that string as an argument. The string argument needs to be mutable so the method can change the string's content by adding the user input.

The `&` indicates that this argument is a *reference*, which gives you a way to let multiple parts of your code access one piece of data without needing to copy that data into memory multiple times. References are a complex feature, and one of Rust's major advantages is how safe and easy it is to use references. You don't need to know a lot of those details to finish this program. For now, all you need to know is that like variables, references are immutable by default. Hence, you need to write `&mut guess` rather than `&guess` to make it mutable. (Chapter 4 will explain references more thoroughly.)

### Handling Potential Failure with the Result Type

We're not quite done with this line of code. Although what we've discussed so far is a single line of text, it's only the first part of the single logical line of code. The second part is this method:

```
.expect("Failed to read line");
```

When you call a method with the `.foo()` syntax, it's often wise to introduce a newline and other whitespace to help break up long lines. We could have written this code as:

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

However, one long line is difficult to read, so it's best to divide it: two lines for two method calls. Now let's discuss what this line does.

As mentioned earlier, `read_line` puts what the user types into the string we're passing it, but it also returns a value—in this case, an `io::Result`. Rust has a number of types named `Result` in its standard library: a generic `Result` as well as specific versions for submodules, such as `io::Result`.

The `Result` types are *enumerations*, often referred to as *enums*. An enumeration is a type that can have a fixed set of values, and those values are called the enum's *variants*. Chapter 6 will cover enums in more detail.

For `Result`, the variants are `Ok` or `Err`. The `Ok` variant indicates the operation was successful, and inside `Ok` is the successfully generated value. The `Err` variant means the operation failed, and `Err` contains information about how or why the operation failed.

The purpose of these `Result` types is to encode error-handling information. Values of the `Result` type, like values of any type, have methods defined on them. An instance of `io::Result` has an `expect` method that you can call. If this instance of `io::Result` is an `Err` value, `expect` will cause the program to crash and display the message that you passed as an argument to `expect`. If the `read_line` method returns an `Err`, it would likely be the result of an error coming from the underlying operating system. If this instance of `io::Result`

is an `Ok` value, `expect` will take the return value that `Ok` is holding and return just that value to you so you can use it. In this case, that value is the number of bytes in what the user entered into standard input.

If you don't call `expect`, the program will compile, but you'll get a warning:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `std::result::Result` which must be used
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(unused_must_use)] on by default
```

Rust warns that you haven't used the `Result` value returned from `read_line`, indicating that the program hasn't handled a possible error.

The right way to suppress the warning is to actually write error handling, but since you just want to crash this program when a problem occurs, you can use `expect`. You'll learn about recovering from errors in Chapter 9.

### Printing Values with println! Placeholders

Aside from the closing curly brackets, there's only one more line to discuss in the code added so far, which is the following:

```
println!("You guessed: {}", guess);
```

This line prints the string we saved the user's input in. The set of curly brackets, {}, is a placeholder: think of {} as little crab pincers that hold a value in place. You can print more than one value using curly brackets: the first set of curly brackets holds the first value listed after the format string, the second set holds the second value, and so on. Printing multiple values in one call to `println!` would look like this:

```
let x = 5;
let y = 10;

println!("x = {} and y = {}", x, y);
```

This code would print `x = 5 and y = 10`.

### Testing the First Part

Let's test the first part of the guessing game. Run it using **`cargo run`**:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev[unoptimized + debug info] target(s) in 1.50 sec
     Running `target/debug/guessing_game`
Guess the number!
```

```
Please input your guess.
6
You guessed: 6
```

At this point, the first part of the game is done: we're getting input from the keyboard and then printing it.

## Generating a Secret Number

Next, we need to generate a secret number that the user will try to guess. The secret number should be different every time so the game is fun to play more than once. Let's use a random number between 1 and 100 so the game isn't too difficult. Rust doesn't yet include random number functionality in its standard library. However, the Rust team does provide a `rand` crate at *https://crates.io/crates/rand/*.

### Using a Crate to Get More Functionality

Remember that a *crate* is a package of Rust code. The project we've been building is a *binary crate*, which is an executable. The `rand` crate is a *library crate*, which contains code intended to be used in other programs.

Cargo's use of external crates is where it really shines. Before we can write code that uses `rand`, we need to modify the *Cargo.toml* file to include the `rand` crate as a dependency. Open that file now and add the following line to the bottom beneath the `[dependencies]` section header that Cargo created for you:

*Cargo.toml*

```
[dependencies]

rand = "0.3.14"
```

In the *Cargo.toml* file, everything that follows a header is part of a section that continues until another section starts. The `[dependencies]` section is where you tell Cargo which external crates your project depends on and which versions of those crates you require. In this case, we'll specify the `rand` crate with the semantic version specifier `0.3.14`. Cargo understands Semantic Versioning (sometimes called *SemVer*), which is a standard for writing version numbers. The number `0.3.14` is actually shorthand for `^0.3.14`, which means "any version that has a public API compatible with version 0.3.14."

Now, without changing any of the code, let's build the project, as shown in Listing 2-2.

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.14
 Downloading libc v0.2.14
   Compiling libc v0.2.14
   Compiling rand v0.3.14
```

```
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev[unoptimized + debug info] target(s) in 1.50 sec
```

*Listing 2-2: The output from running `cargo build` after adding the `rand` crate as a dependency*

You may see different version numbers (but they will all be compatible with the code, thanks to SemVer!), and the lines may be in a different order.

Now that we have an external dependency, Cargo fetches the latest versions of everything from the *registry*, which is a copy of data from *https://crates.io/*. Crates.io is where people in the Rust ecosystem post their open source Rust projects for others to use.

After updating the registry, Cargo checks the `[dependencies]` section and downloads any crates you don't have yet. In this case, although we only listed `rand` as a dependency, Cargo also grabbed a copy of `libc`, because `rand` depends on `libc` to work. After downloading the crates, Rust compiles them and then compiles the project with the dependencies available.

If you immediately run `cargo build` again without making any changes, you won't get any output. Cargo knows it has already downloaded and compiled the dependencies, and you haven't changed anything about them in your *Cargo.toml* file. Cargo also knows that you haven't changed anything about your code, so it doesn't recompile that either. With nothing to do, it simply exits.

If you open the *src/main.rs* file, make a trivial change, and then save it and build again, you'll only see two lines of output:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev[unoptimized + debug info] target(s) in 1.50 sec
```

These lines show Cargo only updates the build with your tiny change to the *src/main.rs* file. Your dependencies haven't changed, so Cargo knows it can reuse what it has already downloaded and compiled for those. It just rebuilds your part of the code.

### Ensuring Reproducible Builds with the Cargo.lock File

Cargo has a mechanism that ensures you can rebuild the same artifact every time you or anyone else builds your code: Cargo will use only the versions of the dependencies you specified until you indicate otherwise. For example, what happens if next week version 0.3.15 of the `rand` crate comes out and contains an important bug fix but also contains a regression that will break your code?

The answer to this problem is the *Cargo.lock* file, which was created the first time you ran `cargo build` and is now in your *guessing_game* directory. When you build a project for the first time, Cargo figures out all the versions of the dependencies that fit the criteria and then writes them to the *Cargo.lock* file. When you build your project in the future, Cargo will see that the *Cargo.lock* file exists and use the versions specified there rather

than doing all the work of figuring out versions again. This lets you have a reproducible build automatically. In other words, your project will remain at 0.3.14 until you explicitly upgrade, thanks to the *Cargo.lock* file.

#### Updating a Crate to Get a New Version

When you *do* want to update a crate, Cargo provides another command, `update`, which will ignore the *Cargo.lock* file and figure out all the latest versions that fit your specifications in *Cargo.toml*. If that works, Cargo will write those versions to the *Cargo.lock* file.

But by default, Cargo will only look for versions larger than 0.3.0 and smaller than 0.4.0. If the `rand` crate has released two new versions, 0.3.15 and 0.4.0, you would see the following if you ran `cargo update`:

```
$ cargo update
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating rand v0.3.14 -> v0.3.15
```

At this point, you would also notice a change in your *Cargo.lock* file noting that the version of the `rand` crate you are now using is 0.3.15.

If you wanted to use `rand` version 0.4.0 or any version in the 0.4.x series, you'd have to update the *Cargo.toml* file to look like this instead:

```
[dependencies]

rand = "0.4.0"
```

The next time you run `cargo build`, Cargo will update the registry of crates available and reevaluate your `rand` requirements according to the new version you have specified.

There's a lot more to say about Cargo and its ecosystem which we'll discuss in Chapter 14, but for now, that's all you need to know. Cargo makes it very easy to reuse libraries, so Rustaceans are able to write smaller projects that are assembled from a number of packages.

### Generating a Random Number

Now that you've added the `rand` crate to Cargo.toml, let's start using `rand`. The next step is to update *src/main.rs*, as shown in Listing 2-3.

*src/main.rs*

```
❶ extern crate rand;

  use std::io;
❷ use rand::Rng;

  fn main() {
      println!("Guess the number!");

   ❸ let secret_number = rand::thread_rng().gen_range(1, 101);

      println!("The secret number is: {}", secret_number);
```

```
    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

*Listing 2-3: Adding code to generate a random number*

First, we add a line that lets Rust know we'll be using the `rand` crate as an external dependency ❶. This also does the equivalent of calling `use rand`, so now we can call anything in the `rand` crate by placing `rand::` before it.

Next, we add another `use` line: `use rand::Rng` ❷. The `Rng` trait defines methods that random number generators implement, and this trait must be in scope for us to use those methods. Chapter 10 will cover traits in detail.

Also, we're adding two more lines in the middle ❸. The `rand::thread_rng` function will give us the particular random number generator that we're going to use: one that is local to the current thread of execution and seeded by the operating system. Next, we call the `gen_range` method on the random number generator. This method is defined by the `Rng` trait that we brought into scope with the `use rand::Rng` statement. The `gen_range` method takes two numbers as arguments and generates a random number between them. It's inclusive on the lower bound but exclusive on the upper bound, so we need to specify `1` and `101` to request a number between 1 and 100.

You won't just know which traits to use and which functions and methods to call from a crate. Instructions for using a crate are in each crate's documentation. Another neat feature of Cargo is that you can run the `cargo doc --open` command, which will build documentation provided by all of your dependencies locally and open it in your browser. If you're interested in other functionality in the `rand` crate, for example, run `cargo doc --open` and click **rand** in the sidebar on the left.

The second line that we added to the code prints the secret number. This is useful while we're developing the program to be able to test it, but we'll delete it from the final version. It's not much of a game if the program prints the answer as soon as it starts!

Try running the program a few times:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev[unoptimized + debug info] target(s) in 1.50 sec
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
```

```
$ cargo run
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

You should get different random numbers, and they should all be numbers between 1 and 100. Great job!

## Comparing the Guess to the Secret Number

Now that we have user input and a random number, we can compare them. That step is shown in Listing 2-4. Note that this code won't compile quite yet, as we will explain.

```
extern crate rand;

use std::io;
❶ use std::cmp::Ordering;
use rand::Rng;

fn main() {

---snip---

    println!("You guessed: {}", guess);

    match❷ guess.cmp(&secret_number)❸ {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

*Listing 2-4: Handling the possible return values of comparing two numbers*

The first new bit here is another `use` statement ❶, bringing a type called `std::cmp::Ordering` into scope from the standard library. Like `Result`, `Ordering` is another enum, but the variants for `Ordering` are `Less`, `Greater`, and `Equal`. These are the three outcomes that are possible when you compare two values.

Then we add five new lines at the bottom that use the `Ordering` type. The `cmp` method ❸ compares two values and can be called on anything that can be compared. It takes a reference to whatever you want to compare with: here it's comparing the `guess` to the `secret_number`. Then it returns a variant of the `Ordering` enum we brought into scope with the `use` statement. We use a `match` expression ❷ to decide what to do next based on which variant of `Ordering` was returned from the call to `cmp` with the values in `guess` and `secret_number`.

A `match` expression is made up of *arms*. An arm consists of a *pattern* and the code that should be run if the value given to the beginning of the `match` expression fits that arm's pattern. Rust takes the value given to `match` and looks through each arm's pattern in turn. The `match` construct and patterns are powerful features in Rust that let you express a variety of situations your code might encounter and make sure to handle them all. These features will be covered in detail in Chapter 6 and Chapter 18, respectively.

Let's walk through an example of what would happen with the `match` expression used here. Say that the user has guessed 50 and the randomly generated secret number this time is 38. When the code compares 50 to 38, the `cmp` method will return `Ordering::Greater`, because 50 is greater than 38. The `match` expression gets the `Ordering::Greater` value and starts checking each arm's pattern. It looks at the first arm's pattern, `Ordering::Less`, and sees that the value `Ordering::Greater` does not match `Ordering::Less`, so it ignores the code in that arm and moves to the next arm. The next arm's pattern, `Ordering::Greater`, *does* match `Ordering::Greater`! The associated code in that arm will execute and print `Too big!` to the screen. The `match` expression ends because it has no need to look at the last arm in this scenario.

However, the code in Listing 2-4 won't compile yet. Let's try it:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^ expected struct `std::string::String`,
found integral variable
   |
   = note: expected type `&std::string::String`
   = note:    found type `&{integer}`

error: aborting due to previous error
Could not compile `guessing_game`.
```

The core of the error states that there are *mismatched types*. Rust has a strong, static type system. However, it also has type inference. When we wrote `let guess = String::new()`, Rust was able to infer that `guess` should be a `String` and didn't make us write the type. The `secret_number`, on the other hand, is a number type. A few number types can have a value between 1 and 100: `i32`, a 32-bit number; `u32`, an unsigned 32-bit number; `i64`, a 64-bit number; as well as others. Rust defaults to an `i32`, which is the type of `secret_number` unless you add type information elsewhere that would cause Rust to infer a different numerical type. The reason for the error here is that Rust cannot compare a string and a number type.

Ultimately, we want to convert the `String` the program reads as input into a real number type so we can compare it numerically to the guess. We can do that by adding the following two lines to the `main` function body:

```
---snip---

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

We create a variable named `guess`. But wait, doesn't the program already have a variable named `guess`? It does, but Rust allows us to *shadow* the previous value of `guess` with a new one. This feature is often used in situations in which you want to convert a value from one type to another type. Shadowing lets us reuse the `guess` variable name rather than forcing us to create two unique variables, such as `guess_str` and `guess`, for example. (Chapter 3 covers shadowing in more detail.)

We bind `guess` to the expression `guess.trim().parse()`. The `guess` in the expression refers to the original `guess` that was a `String` with the input in it. The `trim` method on a `String` instance will eliminate any whitespace at the beginning and end. Although `u32` can contain only numerical characters, the user must press ENTER to satisfy `read_line`. When the user presses ENTER, a newline character is added to the string. For example, if the user types `5` and presses ENTER, `guess` looks like this: `5\n`. The `\n` represents "newline," the result of pressing ENTER. The `trim` method eliminates `\n`, resulting in just `5`.

The `parse` method on strings parses a string into some kind of number. Because this method can parse a variety of number types, we need to tell Rust the exact number type we want by using `let guess: u32`. The colon (`:`) after `guess` tells Rust we'll annotate the variable's type. Rust has a few built-in number types; the `u32` seen here is an unsigned, 32-bit integer. It's a good default choice for a small positive number. You'll learn about other number types in Chapter 3. Additionally, the `u32` annotation in this example program and the comparison with `secret_number` mean that Rust will infer that `secret_number` should be a `u32` type as well. So now the comparison will be between two values of the same type!

The call to `parse` could easily cause an error. If, for example, the string contained `A👍%`, there would be no way to convert that to a number. Because it might fail, the `parse` method returns a `Result` type, much as the `read_line` method does (discussed earlier in "Handling Potential Failure with the Result Type" on page 5). We'll treat this `Result` the same way by using the `expect` method again. If `parse` returns an `Err Result` variant because it couldn't create a number from the string, the `expect` call will crash the game and print the message we give it. If `parse` can successfully convert the string to a number, it will return the `Ok` variant of `Result`, and `expect` will return the number that we want from the `Ok` value.

Let's run the program now!

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev[unoptimized + debug info] target(s) in 1.50 sec
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

Nice! Even though spaces were added before the guess, the program still figured out that the user guessed 76. Run the program a few times to verify the different behavior with different kinds of input: guess the number correctly, guess a number that is too high, and guess a number that is too low.

We have most of the game working now, but the user can make only one guess. Let's change that by adding a loop!

## Allowing Multiple Guesses with Looping

The `loop` keyword creates an infinite loop. We'll add that now to give users more chances at guessing the number:

```
---snip---

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        ---snip---

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
        }
    }
}
```

As you can see, we've moved everything into a loop from the guess input prompt onward. Be sure to indent the lines inside the loop another four spaces each and run the program again. Notice that there is a new problem because the program is doing exactly what we told it to do: ask for another guess forever! It doesn't seem like the user can quit!

The user could always halt the program by using the keyboard shortcut CTRL-C. But there's another way to escape this insatiable monster, as mentioned in the `parse` discussion in "Comparing the Guess to the Secret Number" on page 11: if the user enters a non-number answer, the program will crash. The user can take advantage of that in order to quit, as shown here:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev[unoptimized + debug info] target(s) in 1.50 sec
     Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind:
InvalidDigit }', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/guess` (exit code: 101)
```

Typing `quit` actually quits the game, but so will any other non-number input. However, this is suboptimal to say the least. We want the game to automatically stop when the correct number is guessed.

### Quitting After a Correct Guess

Let's program the game to quit when the user wins by adding a `break` statement:

```
---snip---

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
```

```
                println!("You win!");
                break;
            }
        }
    }
}
```

Adding the `break` line after `You win!` makes the program exit the loop when the user guesses the secret number correctly. Exiting the loop also means exiting the program, because the loop is the last part of `main`.

### Handling Invalid Input

To further refine the game's behavior, rather than crashing the program when the user inputs a non-number, let's make the game ignore a non-number so the user can continue guessing. We can do that by altering the line where `guess` is converted from a `String` to a `u32`:

```
---snip---

io::stdin().read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};

println!("You guessed: {}", guess);

---snip---
```

Switching from an `expect` call to a `match` expression is how you generally move from crashing on an error to handling the error. Remember that `parse` returns a `Result` type and `Result` is an enum that has the variants `Ok` or `Err`. We're using a `match` expression here, as we did with the `Ordering` result of the `cmp` method.

If `parse` is able to successfully turn the string into a number, it will return an `Ok` value that contains the resulting number. That `Ok` value will match the first arm's pattern, and the `match` expression will just return the `num` value that `parse` produced and put inside the `Ok` value. That number will end up right where we want it in the new `guess` variable we're creating.

If `parse` is *not* able to turn the string into a number, it will return an `Err` value that contains more information about the error. The `Err` value does not match the `Ok(num)` pattern in the first `match` arm, but it does match the `Err(_)` pattern in the second arm. The underscore, `_`, is a catchall value; in this example, we're saying we want to match all `Err` values, no matter what information they have inside them. So the program will execute the second arm's code, `continue`, which tells the program to go to the next iteration of the `loop` and ask for another guess. So effectively, the program ignores all errors that `parse` might encounter!

Now everything in the program should work as expected. Let's try it by typing `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev[unoptimized + debug info] target(s) in 1.50 sec
     Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Awesome! With one tiny final tweak, we will finish the guessing game. Recall that the program is still printing the secret number. That worked well for testing, but it ruins the game. Let's delete the `println!` that outputs the secret number. Listing 2-5 shows the final code:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);
```

```
        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

*Listing 2-5: Complete guessing game code*

## Summary

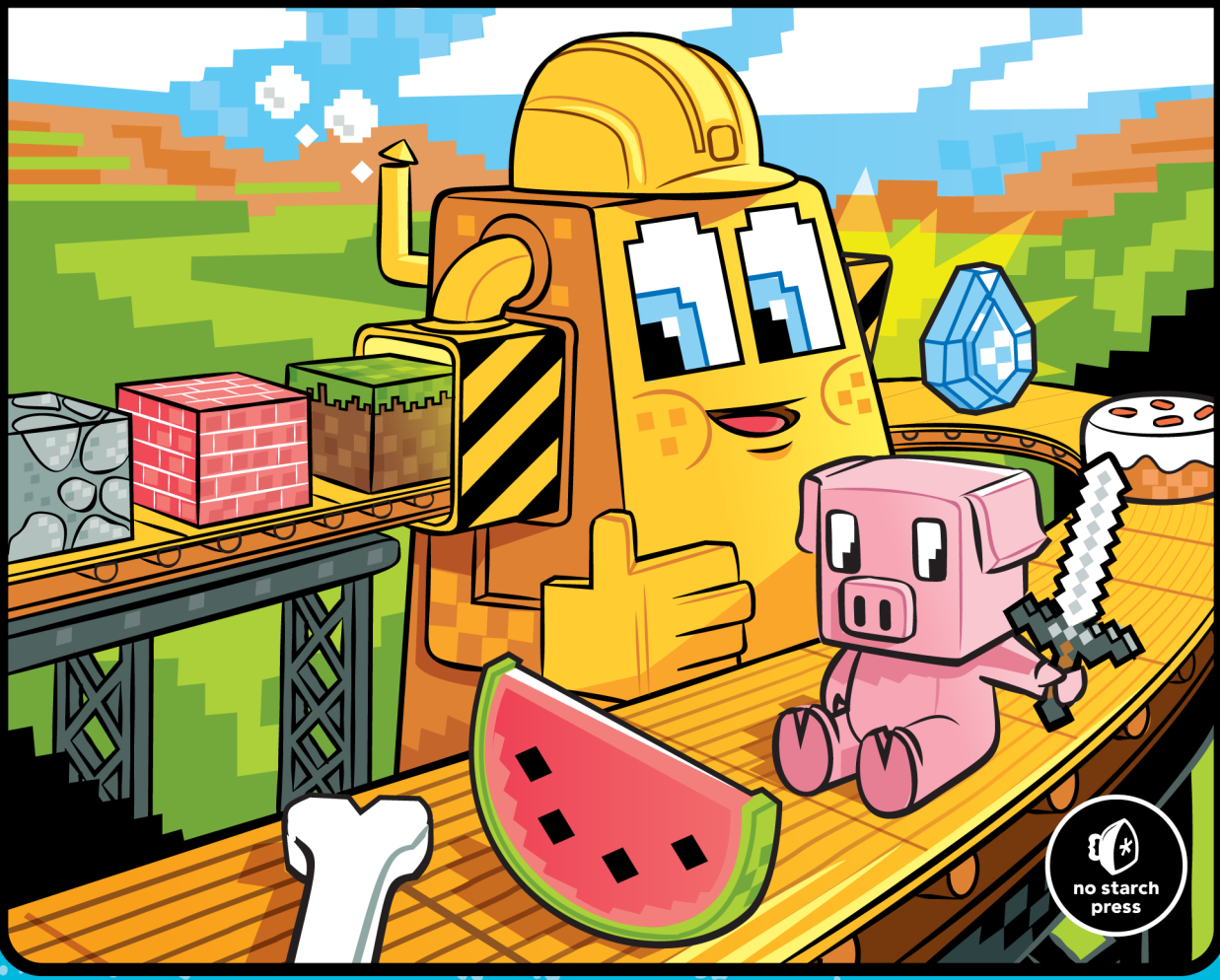
At this point, you've successfully built the guessing game. Congratulations!

This project was a hands-on way to introduce you to many new Rust concepts: `let`, `match`, methods, associated functions, the use of external crates, and more. In the next few chapters, you'll learn about these concepts in more detail. Chapter 3 covers concepts that most programming languages have, such as variables, data types, and functions, and shows how to use them in Rust. Chapter 4 explores ownership, a feature that makes Rust different from other languages. Chapter 5 discusses structs and method syntax, and Chapter 6 explains how enums work.

# CODING WITH MINECRAFT®

## BUILD TALLER, FARM FASTER, MINE DEEPER, AND AUTOMATE THE BORING STUFF

AL SWEIGART



no starch press

# 9

# BUILDING A COBBLESTONE GENERATOR

The most common blocks you'll find as you mine are stone. They become cobblestone when mined, but you can turn them back into stone by smelting them in a furnace. Then you can craft this stone into stone bricks for your buildings' construction materials.

Whew! That's a lot of work in dangerous, dark mines just to get stone bricks. In this chapter, you'll learn how to create a cobblestone generator that will give you infinite cobblestone to work with, and then you'll create a turtle program to automatically mine and smelt that cobblestone into stone. Safe inside your base, you'll have a production line for an endless amount of stone bricks to build with.

## BLUEPRINTS FOR THE COBBLESTONE GENERATOR

Although you can obtain cobblestone by mining it in Minecraft, you can also create it by mixing a stream of water with a stream of lava, which forms a block of cobblestone. You can use this knowledge to build a cobblestone generator that performs the same process to create an unlimited number of cobblestone blocks. Turtles can mine this cobblestone forever because their tools don't wear out.

To create a cobblestone generator, follow the blueprint in Figure 9-1. A generator has three layers of blocks. You'll need one type of block to act as an enclosure for holding the water and lava streams. I used glass, but you can use any nonflammable blocks. You'll need to use three iron to craft a bucket, which you'll then fill from a lava pool. Although you can find lava pools on the surface, they're more commonly found deep underground near bedrock. You can draw water from the rivers, ponds, and oceans on the surface.

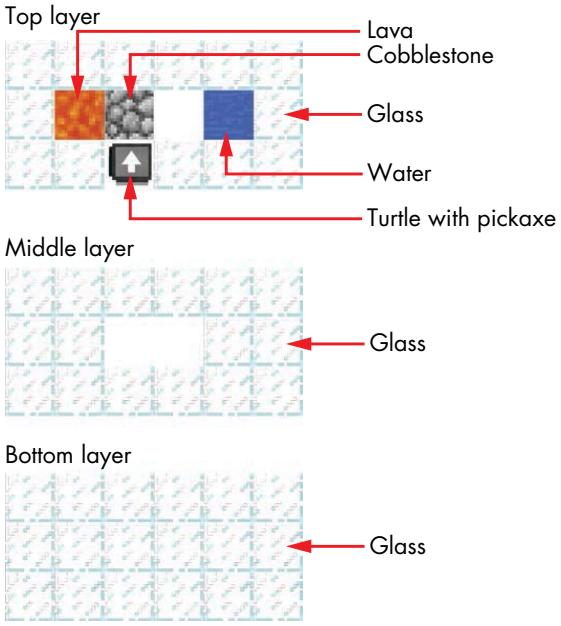



*Figure 9-1: A bird's-eye view of the blueprints for a cobblestone generator*

When placing the lava and water, *be sure to place the lava first*. Otherwise, the water stream will mix directly with the lava block (instead of its stream), turning it into obsidian. You don't need to place the cobblestone block in the blueprint. It will form automatically.

Figure 9-2 shows the completed cobblestone generator.

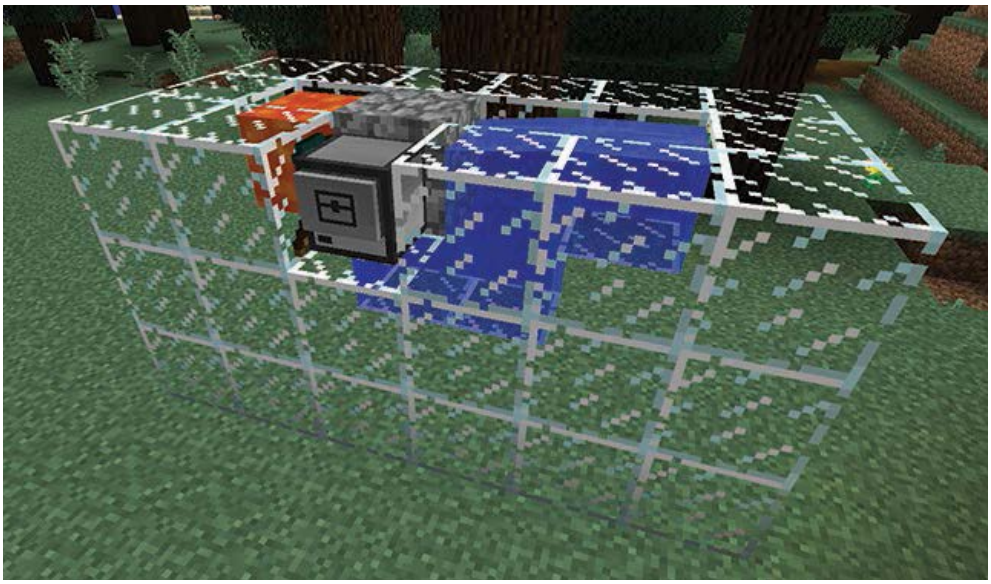


*Figure 9-2: The completed cobblestone generator with a turtle in the empty slot on the top layer*

Whenever the cobblestone block in the generator is mined, it opens space for the lava to flow into to create a new cobblestone block. You could mine the cobblestone an infinite number of times, but you would wear out a lot of pickaxes. By placing a turtle in the open spot on the top layer facing the cobblestone block, you can have the turtle mine the cobblestone blocks forever. Because the turtle doesn't even need to move, it doesn't use any fuel either!

## SETTING UP FURNACES FOR SMELTING THE COBBLESTONE

Even though we now have an infinite supply of cobblestone, we still need to smelt the cobblestone into stone to use it. To do this, we'll create the `cobminer` program, which will make a turtle mine the cobblestone from the generator and then deposit the mined cobblestone into furnaces to smelt it into stone.

Before you create the `cobminer` program, you'll need to do some setup. First, you need to extend the cobblestone generator by adding five furnaces to the middle layer behind the turtle, as shown in Figure 9-3.

The turtle running the new cobblestone miner program will mine until it has a full stack of 64 cobblestones. Then it will move backward over the furnaces, dropping the cobblestone into them. The furnaces will smelt the cobblestone into stone. If all the furnaces are full, the turtle will wait five minutes before trying to drop cobblestone into them again. This entire process will repeat forever.

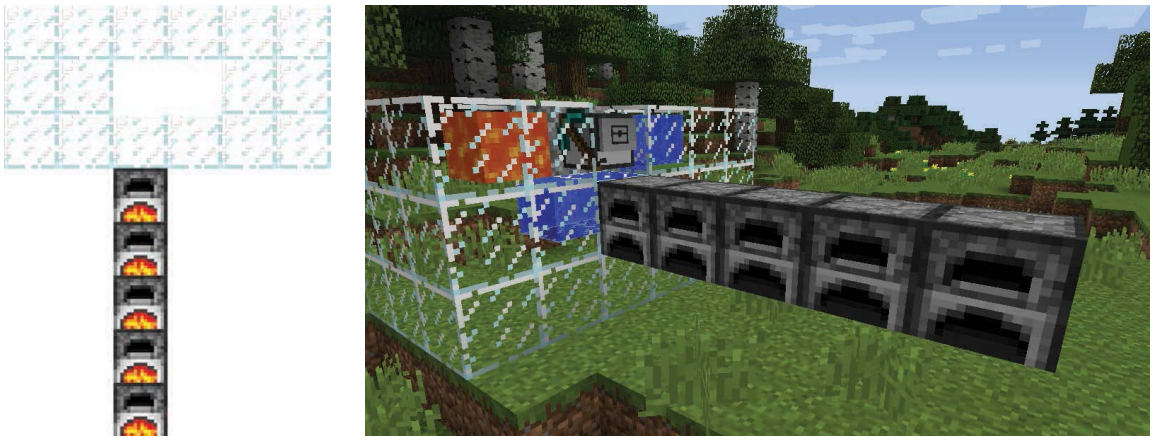


*Figure 9-3: Five furnaces added to the middle layer of the cobblestone generator (left) and the furnaces in the game (right)*

In Chapter 10, we'll create a `brickcrafter` program to run a second turtle. The `brickcrafter` program will pick up the smelted stone from the furnaces and use it to craft stone bricks. The turtle will store these stone brick blocks in a nearby chest for the player.

## WRITING THE COBMINER PROGRAM

To write the `cobminer` program, run **`edit cobminer`** from the command shell and enter the following code:

*cobminer*

```
1. --[[Stone Brick Factory program by Al Sweigart
2. Mines cobblestone from a generator, turtle 1 of 2]]
3.
4. os.loadAPI('hare')  -- load the hare library
5. local numToDrop
6. local NUM_FURNACES = 5
7.
8. print('Starting mining program...')
9. while true do
10.   -- mine cobblestone
11.   if turtle.detect() then
12.     print('Cobblestone detected. Mining...')
13.     turtle.dig()  -- mine cobblestone
14.   else
15.     print('No cobblestone. Sleeping...')
16.     os.sleep(0.5)  -- half second pause
17.   end
18.
19.   -- check for a full stack of cobblestone
20.   hare.selectItem('minecraft:cobblestone')
21.   if turtle.getItemCount() == 64 then
22.     -- check turtle's fuel
23.     if turtle.getFuelLevel() < (2 * NUM_FURNACES) then
24.       error('Turtle needs more fuel!')
25.     end
```

```
26.
27.     -- put cobble in furnaces
28.     print('Dropping off cobblestone...')
29.     for furnacesToFill = NUM_FURNACES, 1, -1 do
30.       turtle.back()  -- move over furnace
31.       numToDrop = math.floor(turtle.getItemCount() / furnacesToFill)
32.       turtle.dropDown(numToDrop)  -- put cobblestone in furnace
33.     end
34.
35.     -- move back to cobblestone generator
36.     for moves = 1, NUM_FURNACES do
37.       turtle.forward()
38.     end
39.
40.     if turtle.getItemCount() > 0 then
41.       print('All furnaces full. Sleeping...')
42.       os.sleep(300)  -- wait for five minutes
43.     end
44.   end
45. end
```

After you've entered these instructions, press CTRL, make sure **[Save]** is selected, and press ENTER. Then quit the editor by pressing CTRL, selecting **[Exit]**, and pressing ENTER. You can also download this program by running **pastebin get YhvSiw7e cobminer**. In addition, you'll need the `hare` module, which you can download by running **pastebin get wwzvaKuW hare**.

## RUNNING THE COBMINER PROGRAM

After you've set up the cobblestone generator with five furnaces, position the turtle facing the cobblestone block and run **cobminer**. The turtle will begin to mine the cobblestone until it has 64 blocks; then it will drop them into the furnaces. Until you write the `brickcrafter` program in Chapter 10, you'll have to manually load fuel into the furnaces and remove the smelted stone blocks from them. To create fuel for the furnaces, smelt the wood blocks from the tree-farming turtles in separate furnaces to produce charcoal for fueling the furnaces. Let's look at each part of the `cobminer` program.

## SETTING UP YOUR PROGRAM AND MAKING A CONSTANT VARIABLE

The first couple of lines of the program contain the usual comment that describes what the program is.

```
1. --[[Stone Brick Factory program by Al Sweigart
2. Mines cobblestone from a generator, turtle 1 of 2]]
3.
4. os.loadAPI('hare')  -- load the hare library
5. local numToDrop
```

Line 4 loads the hare module so the program can call hare.selectItem(). Line 5 declares a variable called numToDrop, which is used later in the program.

Line 6 declares the NUM_FURNACES variable, which contains an integer that represents the number of furnaces that are placed behind the turtle.

```
6. local NUM_FURNACES = 5
```

This program has five furnaces, but you can add more furnaces if you like. If you add more furnaces to your cobblestone generator, set the value of NUM_FURNACES to the new number of furnaces.

The NUM_FURNACES variable is uppercase because it's a *constant* variable, which means its value never changes. Uppercase names for constants are just a convention. Constants are still regular variables. The capitalized name helps remind you that you shouldn't write code that changes the variable. It might seem odd to have a variable whose value never changes, but using constants will make your code easier to understand and will make future changes convenient.

For example, you need to indicate the number of furnaces in your code, but if you use the number instead of NUM_FURNACES in your code and then you later change the number of furnaces, you would have to update your code everywhere 5 is used. When you use a constant like NUM_FURNACES on line 6, you can update your code by just changing the assignment statement. Constants make your code clear and easy to modify.

## MINING THE COBBLESTONE FROM THE GENERATOR

Line 9 begins the program's main while loop, which contains the code to make the turtle mine cobblestone, move over the furnaces, and drop cobblestone into the furnaces.

```
 8. print('Starting mining program...')
 9. while true do
10.   -- mine cobblestone
11.   if turtle.detect() then
12.     print('Cobblestone detected. Mining...')
13.     turtle.dig()  -- mine cobblestone
```

The first part, mining cobblestone, begins on line 11. The turtle.detect() function returns true if a cobblestone block is in front of the turtle. In that case, the program displays Cobblestone detected. Mining... and the call turtle.dig() on line 13 mines the cobblestone.

However, if there is no cobblestone because it was previously mined (and the new cobblestone block hasn't formed yet), turtle.detect() returns false. When the if statement's condition is false, the block of code after the else statement on line 14 runs.

```
14.   else
15.     print('No cobblestone. Sleeping...')
16.     os.sleep(0.5)  -- half second pause
17.   end
```

This code displays `No cobblestone. Sleeping...` and calls `os.sleep(0.5)` to pause the program for half a second so a new cobblestone block has enough time to form. This new cobblestone block will be mined when the program loops around again. When the turtle is done mining cobblestone, it needs to smelt the cobblestone in the furnaces.

## INTERACTING WITH FURNACES

Furnaces have three slots: a *fuel slot* where burnable items like coal power the furnace, an *input slot* for items to be smelted, and an *output slot* where the smelted items remain until the player takes them. The turtle's position next to the furnace determines whether the turtle is putting an item into the furnace as fuel, putting an item in as a block to smelt, or removing the final product. If the turtle is on the side of the furnace, it can drop and take items from the furnace's fuel slot. If a turtle is above a furnace, it can drop and take items from the furnace's input slot to smelt them. If a turtle is below a furnace, it can take smelted items from the furnace's output slot. Figure 9-4 shows these positions.

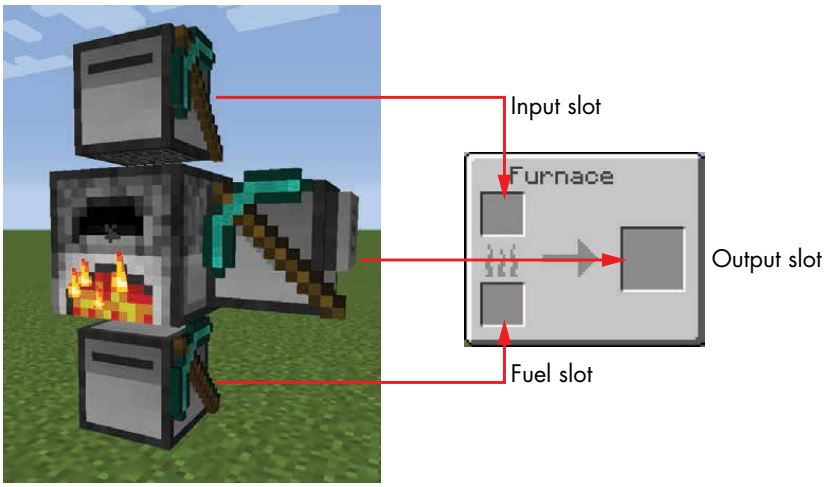



*Figure 9-4: The turtle's position indicates which furnace slot it interacts with.*

Next, the turtle will drop the cobblestone blocks into the furnaces.

### MAKING CODE READABLE WITH CONSTANTS

After checking for a cobblestone block to mine, the program then checks whether the turtle has collected 64 blocks of cobblestone, which is the

maximum an inventory slot can hold. If so, the turtle is ready to drop them into the furnaces behind it after checking that it has enough fuel to travel across the furnaces and back to the cobblestone generator.

```
19.   -- check for a full stack of cobblestone
20.   hare.selectItem('minecraft:cobblestone')
21.   if turtle.getItemCount() == 64 then
22.     -- check turtle's fuel
23.     if turtle.getFuelLevel() < (2 * NUM_FURNACES) then
24.       error('Turtle needs more fuel!')
25.     end
```

The `hare.selectItem()` function on line 20 finds the inventory slot with cobblestone and selects the slot. Line 21 calls `turtle.getItemCount()` to check the number of cobblestone blocks in this slot. If the total is `64` cobblestone blocks, the program calls `turtle.getFuelLevel()` to check the turtle's fuel level.

Line 23 checks whether the turtle's fuel level is less than `2 * NUM_FURNACES`. The reason is that the turtle needs enough fuel to move over each furnace and then move back across each furnace to return to its starting position, as shown in Figure 9-5.

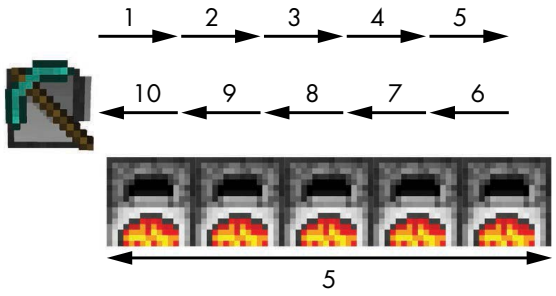



*Figure 9-5: The amount of fuel needed to move over the furnaces and back is twice the number of furnaces.*

If the turtle doesn't have enough fuel, line 24 calls `error()` and displays `Turtle needs more fuel!`. Then line 25 terminates the program.

## DROPPING THE COBBLESTONE INTO THE FURNACES

When the turtle has 64 cobblestone blocks and enough fuel to travel across the furnaces and back, the turtle can move backward and drop off the cobblestone, like in Figure 9-6.

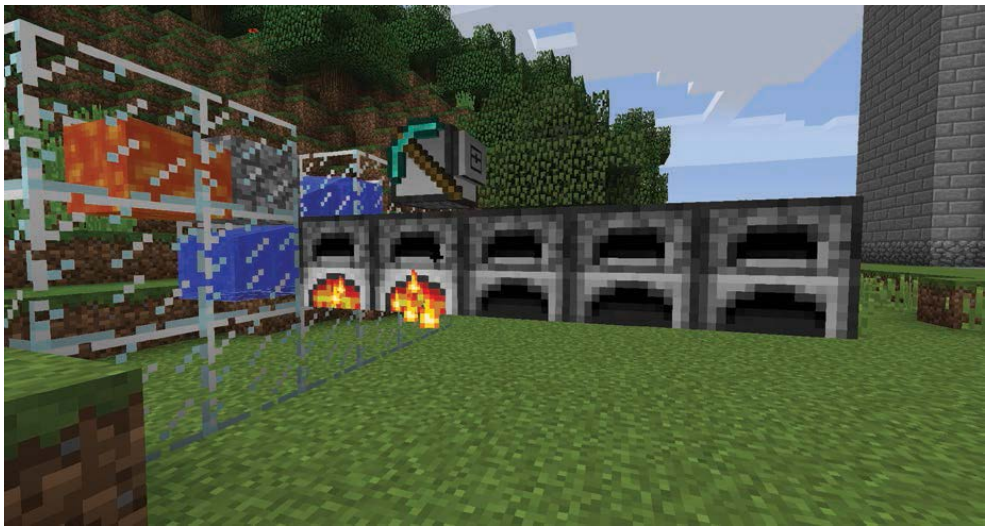


*Figure 9-6: The turtle drops cobblestone blocks into the furnaces.*

The `for` loop on line 29 is slightly different from `for` loops we've used before. A `for` loop can count up, as in `for i = 1, 10 do`, but it can also count in different increments when you include a third number, a *step argument*, to the statement. Instead of adding `1` on each iteration, the `for` loop will add the number indicated in the step argument. If you use a negative number, as in `for i = 10, 1, -1 do`, you can make the loop count down.

```
27.     -- put cobble in furnaces
28.     print('Dropping off cobblestone...')
29.     for furnacesToFill = NUM_FURNACES, 1, -1 do
30.       turtle.back()  -- move over furnace
31.       numToDrop = math.floor(turtle.getItemCount() / furnacesToFill)
32.       turtle.dropDown(numToDrop)  -- put cobblestone in furnace
33.     end
```

The `for` loop on line 29 tells the turtle to move backward `NUM_FURNACES` (or five) times. However, it begins counting at `5` down to `1` instead of `1` up to `5`, so we can use the `for` loop variable, `furnacesToFill`, on line 31 to calculate how many cobblestone blocks to drop into each furnace. This calculation uses the `math.floor()` function. Let's look at how this function works.

## ROUNDING NUMBERS WITH THE MATH.FLOOR() AND MATH.CEIL() FUNCTIONS

The `math.floor()` function rounds down the number it's passed and returns it, whereas the `math.ceil()` function ("ceil" as in "ceiling") rounds up the number it's passed and returns it. Enter the following into the Lua shell to see how these functions work.

```
lua> math.floor(4.2)
4
lua> math.floor(4.9)
4
lua> math.floor(10.5)
10
lua> math.floor(12.0)
12
lua> math.ceil(4.2)
5
lua> math.ceil(4.9)
5
lua> math.ceil(10.5)
11
❶ lua> math.ceil(12.0)
12
```

Passing a value to `math.floor()` results in the number without its decimal point, whereas passing a number to `math.ceil()` rounds up the number to the next number. When you pass `math.ceil()` a number with a decimal value of `0`, it doesn't round up but instead rounds to the closest integer, as you can see at ❶. The functions' roungding helps us evenly distribute the turtle's cobblestone into the furnaces.

## CALCULATING THE COBBLESTONE TO DISTRIBUTE IN EACH FURNACE

In Minecraft, each furnace's input slot can hold up to 64 items to smelt. For efficiency, we want all the furnaces smelting at the same time instead of just one. To calculate how many cobblestone blocks to drop into each furnace, we'll divide the number of cobblestone in the current slot by `NUM_FURNACES`. Because this division operation might not result in a whole number, such as 64 / 5 = 12.8, we'll pass this number to `math.floor()`, which in this case rounds down the number to 12. Then we'll drop that number of cobblestone into each furnace so all the furnaces can smelt at the same time.

But this calculation has a couple of problems. For example, if you have 64 cobblestone blocks and five furnaces, the turtle will drop 12 cobblestone blocks in each furnace and be left with four blocks. Turtles can mine cobblestone quicker than furnaces can smelt them, and each furnace can hold 64 items at most in its input slot. For each furnace that is full and can't accept any more cobblestone, the turtle will be left with the 12 blocks it can drop. In this case, if even one furnace is full, the turtle would be left with 16 blocks of cobblestone! To address this issue, we'll make a different calculation. Let's look at lines 29 to 33 again:

```
29.     for furnacesToFill = NUM_FURNACES, 1, -1 do
30.       turtle.back()  -- move over furnace
31.       numToDrop = math.floor(turtle.getItemCount() / furnacesToFill)
32.       turtle.dropDown(numToDrop)  -- put cobblestone in furnace
33.     end
```

Using numToDrop, line 31 calculates the number of cobblestone blocks to drop in each furnace with numToDrop = math.floor(turtle.getItemCount() / furnacesToFill). Instead of calculating the number of cobblestone blocks to drop once and storing that number in the numToDrop variable, the value of numToDrop is recalculated each time the turtle moves to a new furnace. Table 9-1 shows how numToDrop is calculated on each iteration of the for loop when all the furnaces are empty.

**Table 9-1:** numToDrop Values When All Furnaces Are Empty

| Iteration | math.floor(turtle.getItemCount() / furnacesToFill) | numToDrop | Number of cobblestone dropped into furnace |
|---|---|---|---|
| First | math.floor(64 / 5) | 12 | 12 |
| Second | math.floor(52 / 4) | 13 | 13 |
| Third | math.floor(39 / 3) | 13 | 13 |
| Fourth | math.floor(26 / 2) | 13 | 13 |
| Fifth | math.floor(13 / 1) | 13 | 13 |
| | | | Total: 64 |

However, let's pretend the second furnace is full because the player dropped some of their own mined cobblestone into it. Now no cobblestone can be dropped into the second furnace. But because numToDrop is recalculated on each iteration of the for loop, the code automatically drops more cobblestone into the later furnaces. Table 9-2 shows how numToDrop is calculated on each iteration. Notice that on the second iteration, the number of cobblestone blocks dropped in the furnace is 0 because the second furnace is full.

**Table 9-2:** numToDrop Values When the Second Furnace Is Full

| Iteration | math.floor(turtle.getItemCount() / furnacesToFill) | numToDrop | Number of cobblestone dropped into furnace |
|---|---|---|---|
| First | math.floor(64 / 5) | 12 | 12 |
| Second | math.floor(52 / 4) | 13 | 0 |
| Third | math.floor(52 / 3) | 17 | 17 |
| Fourth | math.floor(35 / 2) | 17 | 17 |
| Fifth | math.floor(18 / 1) | 18 | 18 |
| | | | Total: 64 |

Lines 29 to 33 show that a bit of clever code can make the furnaces work at maximum efficiency. When the for loop has finished, the turtle will be over the last furnace and will need to move back to the cobblestone block.

## MOVING THE COBBLESTONE MINER BACK INTO POSITION

Lines 36 to 38 keep moving the turtle forward until it is in front of the cobblestone block.

```
35.     -- move back to cobblestone generator
36.     for moves = 1, NUM_FURNACES do
37.       turtle.forward()
38.     end
```

At this point, the turtle checks the current slot. Remember, the turtle can mine cobblestone quicker than furnaces can smelt them. It won't be long before all of the furnaces are completely full but the turtle has 64 cobblestone blocks to drop in them. If any cobblestone is still in the turtle's inventory, then all the furnaces are full and the turtle is unable to put this cobblestone in them.

```
40.     if turtle.getItemCount() > 0 then
41.       print('All furnaces full. Sleeping...')
42.       os.sleep(300)  -- wait for 5 minutes
43.     end
44.   end
45. end
```

The `turtle.getItemCount()` returns the number of items in the current slot. If this number is greater than `0` (meaning the turtle still has some cobblestone), line 42 pauses the program for 300 seconds, or five minutes, to give the furnaces more time to smelt the previous cobblestone.

Line 43 ends the `if` statement block on line 40, line 44 ends the `if` statement block on line 21, and line 45 ends the `while` loop on line 9. Finally, the execution loops back to line 9 and the turtle continues mining cobblestone and filling the furnaces until it runs out of fuel.

As with the tree-farming program in Chapter 8, you can scale your cobblestone production by building multiple cobblestone generators. You can also add more furnaces behind the turtle and change the `NUM_FURNACES` constant. (Five or six furnaces are plenty to smelt cobblestone. Otherwise, your turtle won't be able to mine fast enough to keep up with the furnaces!)

## SUMMARY

In this chapter, you learned how to build a cobblestone generator that mixes lava and water streams to produce an endless supply of cobblestone blocks for your turtle to mine, and you used the `cobminer` program to make the turtle mine these cobblestone blocks and drop them into furnaces behind the turtle. You also learned about constants, which are variables that don't change their values and which make your code more readable. In addition, you learned about the step argument in `for` loops, which lets you

create `for` loops that count down instead of up. Finally, you learned how the `math.floor()` and `math.ceil()` functions can round a number down and up, respectively.

In Chapter 10, we'll use the `cobminer` program to create a stone brick factory, which is a two-turtle operation. We'll write a program that will instruct another turtle to take smelted stone blocks out of the furnaces and craft them into stone brick using the `brickcrafter` program.

# CRACKING CODES WITH PYTHON

## AN INTRODUCTION TO *BUILDING* AND *BREAKING* *CIPHERS*

AL SWEIGART

no starch press

# 5

## THE CAESAR CIPHER

*"BIG BROTHER IS WATCHING YOU."*
*—George Orwell*, Nineteen Eighty-Four

In Chapter 1, we used a cipher wheel and a chart of letters and numbers to implement the Caesar cipher. In this chapter, we'll implement the Caesar cipher in a computer program.

The reverse cipher we made in Chapter 4 always encrypts the same way. But the Caesar cipher uses keys, which encrypt the message differently depending on which key is used. The keys for the Caesar cipher are the integers from 0 to 25. Even if a cryptanalyst knows the Caesar cipher was used, that alone doesn't give them enough information to break the cipher. They must also know the key.

**TOPICS COVERED IN THIS CHAPTER**

- The `import` statement
- Constants
- `for` loops
- `if`, `else`, and `elif` statements
- The `in` and `not in` operators
- The `find()` string method

## Source Code for the Caesar Cipher Program

Enter the following code into the file editor and save it as *caesarCipher.py*. Then download the *pyperclip.py* module from *https://www.nostarch.com/crackingcodes/* and place it in the same directory (that is, the same folder) as the file *caesarCipher.py*. This module will be imported by *caesarCipher.py*; we'll discuss this in more detail in "Importing Modules and Setting Up Variables" on page 56.

When you're finished setting up the files, press F5 to run the program. If you run into any errors or problems with your code, you can compare it to the code in the book using the online diff tool at *https://www.nostarch.com/crackingcodes/*.

*caesarCipher.py*

```
 1. # Caesar Cipher
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
 5.
 6. # The string to be encrypted/decrypted:
 7. message = 'This is my secret message.'
 8.
 9. # The encryption/decryption key:
10. key = 13
11.
12. # Whether the program encrypts or decrypts:
13. mode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
14.
15. # Every possible symbol that can be encrypted:
16. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
17.
18. # Store the encrypted/decrypted form of the message:
19. translated = ''
20.
```

```
21. for symbol in message:
22.     # Note: Only symbols in the SYMBOLS string can be
          encrypted/decrypted.
23.     if symbol in SYMBOLS:
24.         symbolIndex = SYMBOLS.find(symbol)
25.
26.         # Perform encryption/decryption:
27.         if mode == 'encrypt':
28.             translatedIndex = symbolIndex + key
29.         elif mode == 'decrypt':
30.             translatedIndex = symbolIndex - key
31.
32.         # Handle wraparound, if needed:
33.         if translatedIndex >= len(SYMBOLS):
34.             translatedIndex = translatedIndex - len(SYMBOLS)
35.         elif translatedIndex < 0:
36.             translatedIndex = translatedIndex + len(SYMBOLS)
37.
38.         translated = translated + SYMBOLS[translatedIndex]
39.     else:
40.         # Append the symbol without encrypting/decrypting:
41.         translated = translated + symbol
42.
43. # Output the translated string:
44. print(translated)
45. pyperclip.copy(translated)
```

## Sample Run of the Caesar Cipher Program

When you run the *caesarCipher.py* program, the output looks like this:

```
guv6Jv6Jz!J6rp5r7Jzr66ntrM
```

The output is the string `'This is my secret message.'` encrypted with the Caesar cipher using a key of `13`. The Caesar cipher program you just ran automatically copies this encrypted string to the clipboard so you can paste it in an email or text file. As a result, you can easily send the encrypted output from the program to another person.

You might see the following error message when you run the program:

```
Traceback (most recent call last):
  File "C:\caesarCipher.py", line 4, in <module>
    import pyperclip
ImportError: No module named pyperclip
```

If so, you probably haven't downloaded the *pyperclip.py* module into the right folder. If you confirm that *pyperclip.py* is in the folder with *caesarCipher.py* but still can't get the module to work, just comment out the code on lines 4 and 45 (which have the text `pyperclip` in them) from the *caesarCipher.py* program by placing a `#` in front of them. This makes Python ignore the code

that depends on the *pyperclip.py* module and should allow the program to run successfully. Note that if you comment out that code, the encrypted or decrypted text won't be copied to the clipboard at the end of the program. You can also comment out the `pyperclip` code from the programs in future chapters, which will remove the copy-to-clipboard functionality from those programs, too.

To decrypt the message, just paste the output text as the new value stored in the `message` variable on line 7. Then change the assignment statement on line 13 to store the string `'decrypt'` in the variable `mode`:

```
 6. # The string to be encrypted/decrypted:
 7. message = 'guv6Jv6Jz!J6rp5r7Jzr66ntrM'
 8.
 9. # The encryption/decryption key:
10. key = 13
11.
12. # Whether the program encrypts or decrypts:
13. mode = 'decrypt' # Set to either 'encrypt' or 'decrypt'.
```

When you run the program now, the output looks like this:

```
This is my secret message.
```

## Importing Modules and Setting Up Variables

Although Python includes many built-in functions, some functions exist in separate programs called modules. *Modules* are Python programs that contain additional functions that your program can use. We import modules with the appropriately named `import` statement, which consists of the `import` keyword followed by the module name.

Line 4 contains an `import` statement:

```
 1. # Caesar Cipher
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
```

In this case, we're importing a module named `pyperclip` so we can call the `pyperclip.copy()` function later in this program. The `pyperclip.copy()` function will automatically copy strings to your computer's clipboard so you can conveniently paste them into other programs.

The next few lines in *caesarCipher.py* set three variables:

```
 6. # The string to be encrypted/decrypted:
 7. message = 'This is my secret message.'
 8.
 9. # The encryption/decryption key:
10. key = 13
```

```
11.
12. # Whether the program encrypts or decrypts:
13. mode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
```

The `message` variable stores the string to be encrypted or decrypted, and the `key` variable stores the integer of the encryption key. The `mode` variable stores either the string `'encrypt'`, which makes code later in the program encrypt the string in `message`, or `'decrypt'`, which makes the program decrypt rather than encrypt.

## Constants and Variables

*Constants* are variables whose values shouldn't be changed when the program runs. For example, the Caesar cipher program needs a string that contains every possible character that can be encrypted with this Caesar cipher. Because that string shouldn't change, we store it in the constant variable named `SYMBOLS` in line 16:

```
15. # Every possible symbol that can be encrypted:
16. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
```

*Symbol* is a common term used in cryptography for a single character that a cipher can encrypt or decrypt. A *symbol set* is every possible symbol a cipher is set up to encrypt or decrypt. Because we'll use the symbol set many times in this program, and because we don't want to type the full string value each time it appears in the program (we might make typos, which would cause errors), we use a constant variable to store the symbol set. We enter the code for the string value once and place it in the `SYMBOLS` constant.

Note that `SYMBOLS` is in all uppercase letters, which is the naming convention for constants. Although we *could* change `SYMBOLS` just like any other variable, the all uppercase name reminds the programmer not to write code that does so.

As with all conventions, we don't *have* to follow this one. But doing so makes it easier for other programmers to understand how these variables are used. (It can even help you when you're looking at your own code later.)

On line 19, the program stores a blank string in a variable named `translated` that will later store the encrypted or decrypted message:

```
18. # Store the encrypted/decrypted form of the message:
19. translated = ''
```

Just as in the reverse cipher in Chapter 5, by the end of the program, the `translated` variable will contain the completely encrypted (or decrypted) message. But for now it starts as a blank string.

## The for Loop Statement

At line 21, we use a type of loop called a `for` loop:

```
21. for symbol in message:
```

Recall that a `while` loop will loop as long as a certain condition is `True`. The `for` loop has a slightly different purpose and doesn't have a condition like the `while` loop. Instead, it loops over a string or a group of values. Figure 5-1 shows the six parts of a `for` loop.

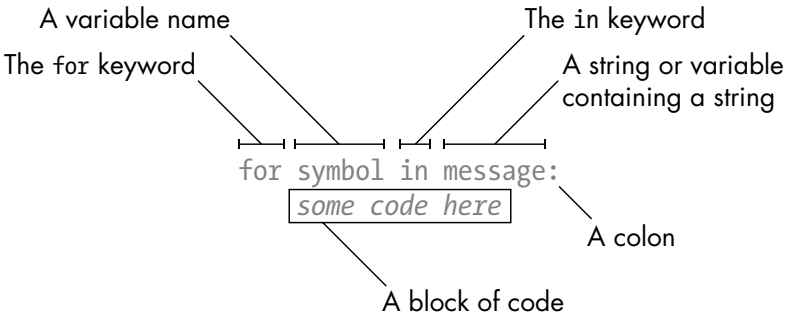



*Figure 5-1: The six parts of a for loop statement*

Each time the program execution goes through the loop (that is, on each iteration through the loop) the variable in the `for` statement (which in line 21 is `symbol`) takes on the value of the next character in the variable containing a string (which in this case is `message`). The `for` statement is similar to an assignment statement because the variable is created and assigned a value except the `for` statement cycles through different values to assign the variable.

### An Example for Loop

For example, type the following into the interactive shell. Note that after you type the first line, the `>>>` prompt will disappear (represented in our code as `...`) because the shell is expecting a block of code after the `for` statement's colon. In the interactive shell, the block will end when you enter a blank line:

```
>>> for letter in 'Howdy':
...     print('The letter is ' + letter)
...
The letter is H
The letter is o
The letter is w
The letter is d
The letter is y
```

This code loops over each character in the string `'Howdy'`. When it does, the variable `letter` takes on the value of each character in `'Howdy'` one at a time in order. To see this in action, we've written code in the loop that prints the value of `letter` for each iteration.

### A while Loop Equivalent of a for Loop

The `for` loop is very similar to the `while` loop, but when you only need to iterate over characters in a string, using a `for` loop is more efficient. You could make a `while` loop act like a `for` loop by writing a bit more code:

```
❶ >>> i = 0
❷ >>> while i < len('Howdy'):
❸ ...     letter = 'Howdy'[i]
❹ ...     print('The letter is ' + letter)
❺ ...     i = i + 1
...
The letter is H
The letter is o
The letter is w
The letter is d
The letter is y
```

Notice that this `while` loop works the same as the `for` loop but is not as short and simple as the `for` loop. First, we set a new variable `i` to `0` before the `while` statement ❶. This statement has a condition that will evaluate to `True` as long as the variable `i` is less than the length of the string `'Howdy'` ❷. Because `i` is an integer and only keeps track of the current position in the string, we need to declare a separate `letter` variable to hold the character in the string at the `i` position ❸. Then we can print the current value of `letter` to get the same output as the `for` loop ❹. When the code is finished executing, we need to increment `i` by adding `1` to it to move to the next position ❺.

To understand lines 23 and 24 in *caesarCipher.py*, you need to learn about the `if`, `elif`, and `else` statements, the `in` and `not in` operators, and the `find()` string method. We'll look at these in the following sections.

## The if Statement

Line 23 in the Caesar cipher has another kind of Python instruction—the `if` statement:

```
23.     if symbol in SYMBOLS:
```

You can read an `if` statement as, "If this condition is `True`, execute the code in the following block. Otherwise, if it is `False`, skip the block." An `if` statement is formatted using the keyword `if` followed by a condition, followed by a colon (`:`). The code to execute is indented in a block just as with loops.

### An Example if Statement

Let's try an example of an `if` statement. Open a new file editor window, enter the following code, and save it as *checkPw.py*:

*checkPw.py*

```
print('Enter your password.')
❶ typedPassword = input()
❷ if typedPassword == 'swordfish':
❸     print('Access Granted')
❹ print('Done')
```

When you run this program, it displays the text `Enter your password.` and lets the user type in a password. The password is then stored in the variable `typedPassword` ❶. Next, the `if` statement checks whether the password is equal to the string `'swordfish'` ❷. If it is, the execution moves inside the block following the `if` statement to display the text `Access Granted` to the user ❸; otherwise, if `typedPassword` isn't equal to `'swordfish'`, the execution skips the `if` statement's block. Either way, the execution continues on to the code after the `if` block to display `Done` ❹.

### The else Statement

Often, we want to test a condition and execute one block of code if the condition is `True` and another block of code if it's `False`. We can use an `else` statement after an `if` statement's block, and the `else` statement's block of code will be executed if the `if` statement's condition is `False`. For an `else` statement, you just write the keyword `else` and a colon (`:`). It doesn't need a condition because it will be run if the `if` statement's condition isn't true. You can read the code as, "If this condition is `True`, execute this block, or else, if it is `False`, execute this other block."

Modify the *checkPw.py* program to look like the following (the new lines are in bold):

*checkPw.py*

```
print('Enter your password.')
typedPassword = input()
❶ if typedPassword == 'swordfish':
    print('Access Granted')
else:
❷     print('Access Denied')
❸ print('Done')
```

This version of the program works almost the same as the previous version. The text `Access Granted` will still display if the `if` statement's condition is `True` ❶. But now if the user types something other than `swordfish`, the `if` statement's condition will be `False`, causing the execution to enter the `else` statement's block and display `Access Denied` ❷. Either way, the execution will still continue and display `Done` ❸.

### The elif Statement

Another statement, called the `elif` statement, can also be paired with `if`. Like an `if` statement, it has a condition. Like an `else` statement, it follows an `if` (or another `elif`) statement and executes if the previous `if` (or `elif`) statement's condition is `False`. You can read `if`, `elif`, and `else` statements as, "If this condition is `True`, run this block. Or else, check if this next condition is `True`. Or else, just run this last block." Any number of `elif` statements can follow an `if` statement. Modify the *checkPw.py* program again to make it look like the following:

*checkPw.py*

```
print('Enter your password.')
typedPassword = input()
❶ if typedPassword == 'swordfish':
❷     print('Access Granted')
❸ elif typedPassword == 'mary':
      print('Hint: the password is a fish.')
❹ elif typedPassword == '12345':
      print('That is a really obvious password.')
else:
      print('Access Denied')
print('Done')
```

This code contains four blocks for the `if`, `elif`, and `else` statements. If the user enters `12345`, then `typedPassword == 'swordfish'` evaluates to `False` ❶, so the first block with `print('Access Granted')` ❷ is skipped. The execution next checks the `typedPassword == 'mary'` condition, which also evaluates to `False` ❸, so the second block is also skipped. The `typedPassword == '12345'` condition is `True` ❹, so the execution enters the block following this `elif` statement to run the code `print('That is a really obvious password.')` and skips any remaining `elif` and `else` statements. *Notice that one and only one of these blocks will be executed.*

You can have zero or more `elif` statements following an `if` statement. You can have zero or one but not multiple `else` statements, and the `else` statement always comes last because it only executes if none of the conditions evaluate to `True`. The first statement with a `True` condition has its block executed. The rest of the conditions (even if they're also `True`) aren't checked.

## The in and not in Operators

Line 23 in *caesarCipher.py* also uses the `in` operator:

```
23.     if symbol in SYMBOLS:
```

An `in` operator can connect two strings, and it will evaluate to `True` if the first string is inside the second string or evaluate to `False` if not. The `in`

operator can also be paired with `not`, which will do the opposite. Enter the following into the interactive shell:

```
>>> 'hello' in 'hello world!'
True
>>> 'hello' not in 'hello world!'
False
>>> 'ello' in 'hello world!'
True
❶ >>> 'HELLO' in 'hello world!'
False
❷ >>> '' in 'Hello'
True
```

Notice that the `in` and `not in` operators are case sensitive ❶. Also, a blank string is always considered to be in any other string ❷.

Expressions using the `in` and `not in` operators are handy to use as conditions of `if` statements to execute some code if a string exists inside another string.

Returning to *caesarCipher.py*, line 23 checks whether the string in `symbol` (which the `for` loop on line 21 set to a single character from the `message` string) is in the `SYMBOLS` string (the symbol set of all characters that can be encrypted or decrypted by this cipher program). If `symbol` is in `SYMBOLS`, the execution enters the block that follows starting on line 24. If it isn't, the execution skips this block and instead enters the block following line 39's `else` statement. The cipher program needs to run different code depending on whether the symbol is in the symbol set.

## The find() String Method

Line 24 finds the index in the `SYMBOLS` string where `symbol` is:

```
24.         symbolIndex = SYMBOLS.find(symbol)
```

This code includes a method call. *Methods* are just like functions except they're attached to a value with a period (or in line 24, a variable containing a value). The name of this method is `find()`, and it's being called on the string value stored in `SYMBOLS`.

Most data types (such as strings) have methods. The `find()` method takes one string argument and returns the integer index of where the argument appears in the method's string. Enter the following into the interactive shell:

```
>>> 'hello'.find('e')
1
>>> 'hello'.find('o')
4
>>> spam = 'hello'
>>> spam.find('h')
❶ 0
```

You can use the `find()` method on either a string or a variable containing a string value. Remember that indexing in Python starts with `0`, so when the index returned by `find()` is for the first character in the string, a `0` is returned ❶.

If the string argument can't be found, the `find()` method returns the integer `-1`. Enter the following into the interactive shell:

```
>>> 'hello'.find('x')
-1
❶ >>> 'hello'.find('H')
-1
```

Notice that the `find()` method is also case sensitive ❶.

The string you pass as an argument to `find()` can be more than one character. The integer that `find()` returns will be the index of the first character where the argument is found. Enter the following into the interactive shell:

```
>>> 'hello'.find('ello')
1
>>> 'hello'.find('lo')
3
>>> 'hello hello'.find('e')
1
```

The `find()` string method is like a more specific version of using the `in` operator. It not only tells you whether a string exists in another string but also tells you where.

## Encrypting and Decrypting Symbols

Now that you understand `if`, `elif`, and `else` statements; the `in` operator; and the `find()` string method, it will be easier to understand how the rest of the Caesar cipher program works.

The cipher program can only encrypt or decrypt symbols that are in the symbol set:

```
23.     if symbol in SYMBOLS:
24.         symbolIndex = SYMBOLS.find(symbol)
```

So before running the code on line 24, the program must figure out whether `symbol` is in the symbol set. Then it can find the index in `SYMBOLS` where `symbol` is located. The index returned by the `find()` call is stored in `symbolIndex`.

Now that we have the current symbol's index stored in `symbolIndex`, we can do the encryption or decryption math on it. The Caesar cipher adds the key number to the symbol's index to encrypt it or subtracts the key number

from the symbol's index to decrypt it. This value is stored in `translatedIndex` because it will be the index in `SYMBOLS` of the translated symbol.

*caesarCipher.py*

```
26.         # Perform encryption/decryption:
27.         if mode == 'encrypt':
28.             translatedIndex = symbolIndex + key
29.         elif mode == 'decrypt':
30.             translatedIndex = symbolIndex - key
```

The `mode` variable contains a string that tells the program whether it should be encrypting or decrypting. If this string is `'encrypt'`, then the condition for line 27's `if` statement will be `True`, and line 28 will be executed to add the `key` to `symbolIndex` (and the block after the `elif` statement will be skipped). Otherwise, if `mode` is `'decrypt'`, then line 30 is executed to subtract the `key`.

### Handling Wraparound

When we were implementing the Caesar cipher with paper and pencil in Chapter 1, sometimes adding or subtracting the key would result in a number greater than or equal to the size of the symbol set or less than zero. In those cases, we have to add or subtract the length of the symbol set so that it will "wrap around," or return to the beginning or end of the symbol set. We can use the code `len(SYMBOLS)` to do this, which returns `66`, the length of the `SYMBOLS` string. Lines 33 to 36 handle this wraparound in the cipher program.

```
32.         # Handle wraparound, if needed:
33.         if translatedIndex >= len(SYMBOLS):
34.             translatedIndex = translatedIndex - len(SYMBOLS)
35.         elif translatedIndex < 0:
36.             translatedIndex = translatedIndex + len(SYMBOLS)
```

If `translatedIndex` is greater than or equal to `66`, the condition on line 33 is `True` and line 34 is executed (and the `elif` statement on line 35 is skipped). Subtracting the length of `SYMBOLS` from `translatedIndex` points the index of the variable back to the beginning of the `SYMBOLS` string. Otherwise, Python will check whether `translatedIndex` is less than `0`. If that condition is `True`, line 36 is executed, and `translatedIndex` wraps around to the end of the `SYMBOLS` string.

You might be wondering why we didn't just use the integer value `66` directly instead of `len(SYMBOLS)`. By using `len(SYMBOLS)` instead of `66`, we can add to or remove symbols from `SYMBOLS` and the rest of the code will still work.

Now that you have the index of the translated symbol in `translatedIndex`, `SYMBOLS[translatedIndex]` will evaluate to the translated symbol. Line 38 adds this encrypted/decrypted symbol to the end of the `translated` string using string concatenation:

```
38.         translated = translated + SYMBOLS[translatedIndex]
```

Eventually, the `translated` string will be the whole encoded or decoded message.

### *Handling Symbols Outside of the Symbol Set*

The `message` string might contain characters that are not in the `SYMBOLS` string. These characters are outside of the cipher program's symbol set and can't be encrypted or decrypted. Instead, they will just be appended to the `translated` string as is, which happens in lines 39 to 41:

```
39.     else:
40.         # Append the symbol without encrypting/decrypting:
41.         translated = translated + symbol
```

The `else` statement on line 39 has four spaces of indentation. If you look at the indentation of the lines above, you'll see that it's paired with the `if` statement on line 23. Although there's a lot of code in between this `if` and `else` statement, it all belongs in the same block of code.

If line 23's `if` statement's condition were `False`, the block would be skipped, and the program execution would enter the `else` statement's block starting at line 41. This `else` block has just one line in it. It adds the unchanged `symbol` string to the end of `translated`. As a result, symbols outside of the symbol set, such as `'%'` or `'('`, are added to the translated string without being encrypted or decrypted.

## Displaying and Copying the Translated String

Line 43 has no indentation, which means it's the first line after the block that started on line 21 (the `for` loop's block). By the time the program execution reaches line 44, it has looped through each character in the `message` string, encrypted (or decrypted) the characters, and added them to `translated`:

```
43. # Output the translated string:
44. print(translated)
45. pyperclip.copy(translated)
```

Line 44 calls the `print()` function to display the `translated` string on the screen. Notice that this is the only `print()` call in the entire program. The computer does a lot of work encrypting every letter in `message`, handling wraparound, and handling non-letter characters. But the user doesn't need to see this. The user just needs to see the final string in `translated`.

Line 45 calls `copy()`, which takes one string argument and copies it to the clipboard. Because `copy()` is a function in the `pyperclip` module, we must tell Python this by putting `pyperclip.` in front of the function name. If we type `copy(translated)` instead of `pyperclip.copy(translated)`, Python will give us an error message because it won't be able to find the function.

Python will also give an error message if you forget the `import pyperclip` line (line 4) before trying to call `pyperclip.copy()`.

That's the entire Caesar cipher program. When you run it, notice how your computer can execute the entire program and encrypt the string in less than a second. Even if you enter a very long string to store in the `message`

variable, your computer can encrypt or decrypt the message within a second or two. Compare this to the several minutes it would take to do this with a cipher wheel. The program even automatically copies the encrypted text to the clipboard so the user can simply paste it into an email to send to someone.

## Encrypting Other Symbols

One problem with the Caesar cipher that we've implemented is that it can't encrypt characters outside its symbol set. For example, if you encrypt the string `'Be sure to bring the $$$.'` with the key `20`, the message will encrypt to `'VyQ?A!yQ.9Qv!381Q.2yQ$$$T'`. This encrypted message doesn't hide that you are referring to $$$. However, we can modify the program to encrypt other symbols.

By changing the string that is stored in `SYMBOLS` to include more characters, the program will encrypt them as well, because on line 23, the condition `symbol in SYMBOLS` will be `True`. The value of `symbolIndex` will be the index of `symbol` in this new, larger `SYMBOLS` constant variable. The "wraparound" will need to add or subtract the number of characters in this new string, but that's already handled because we use `len(SYMBOLS)` instead of typing `66` directly into the code (which is why we programmed it this way).

For example, you could expand line 16 to be:

```
SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
  67890 !?.`~@#$%^&*()_+-=[]{}|;:<>,/'
```

Keep in mind that a message must be encrypted and decrypted with the same symbol set to work.

## Summary

You've learned several programming concepts and read through quite a few chapters to get to this point, but now you have a program that implements a secret cipher. And more important, you understand how this code works.

Modules are Python programs that contain useful functions. To use these functions, you must first import them using an `import` statement. To call functions in an imported module, put the module name and a period before the function name, like so: *`module.function`*`()`.

Constant variables are written in uppercase letters by convention. These variables are not meant to have their values changed (although nothing prevents the programmer from writing code that does so). Constants are helpful because they give a "name" to specific values in your program.

Methods are functions that are attached to a value of a certain data type. The `find()` string method returns an integer of the position of the string argument passed to it inside the string it is called on.

You learned about several new ways to manipulate which lines of code run and how many times each line runs. A `for` loop iterates over all the characters in a string value, setting a variable to each character on each iteration. The `if`, `elif`, and `else` statements execute blocks of code based on whether a condition is `True` or `False`.

The `in` and `not in` operators check whether one string is or isn't in another string and evaluate to `True` or `False` accordingly.

Knowing how to program gives you the ability to write down a process like encrypting or decrypting with the Caesar cipher in a language that a computer can understand. And once the computer understands how to execute the process, it can do it much faster than any human can and with no mistakes (unless mistakes are in your programming). Although this is an incredibly useful skill, it turns out the Caesar cipher can easily be broken by someone who knows how to program. In Chapter 6, you'll use the skills you've learned to write a Caesar cipher hacker so you can read ciphertext that other people have encrypted. Let's move on and learn how to hack encryption.

### PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Using *caesarCipher.py*, encrypt the following sentences with the given keys:
   a. `'"You can show black is white by argument," said Filby, "but you will never convince me."'` with key `8`
   b. `'1234567890'` with key `21`
2. Using *caesarCipher.py*, decrypt the following ciphertexts with the given keys:
   a. `'Kv?uqwpfu?rncwukdng?gpqwijB'` with key `2`
   b. `'XCBSw88S18A1S 2SB41SE .8zSEwAS5OD5A5x81V'` with key `22`
3. Which Python instruction would import a module named *watermelon.py*?
4. What do the following pieces of code display on the screen?
   a.

```
spam = 'foo'
for i in spam:
    spam = spam + i
print(spam)
```

*(continued)*

b.

```
if 10 < 5:
    print('Hello')
elif False:
    print('Alice')
elif 5 != 5:
    print('Bob')
else:
    print('Goodbye')
```

c.

```
print('f' not in 'foo')
```

d.

```
print('foo' in 'f')
```

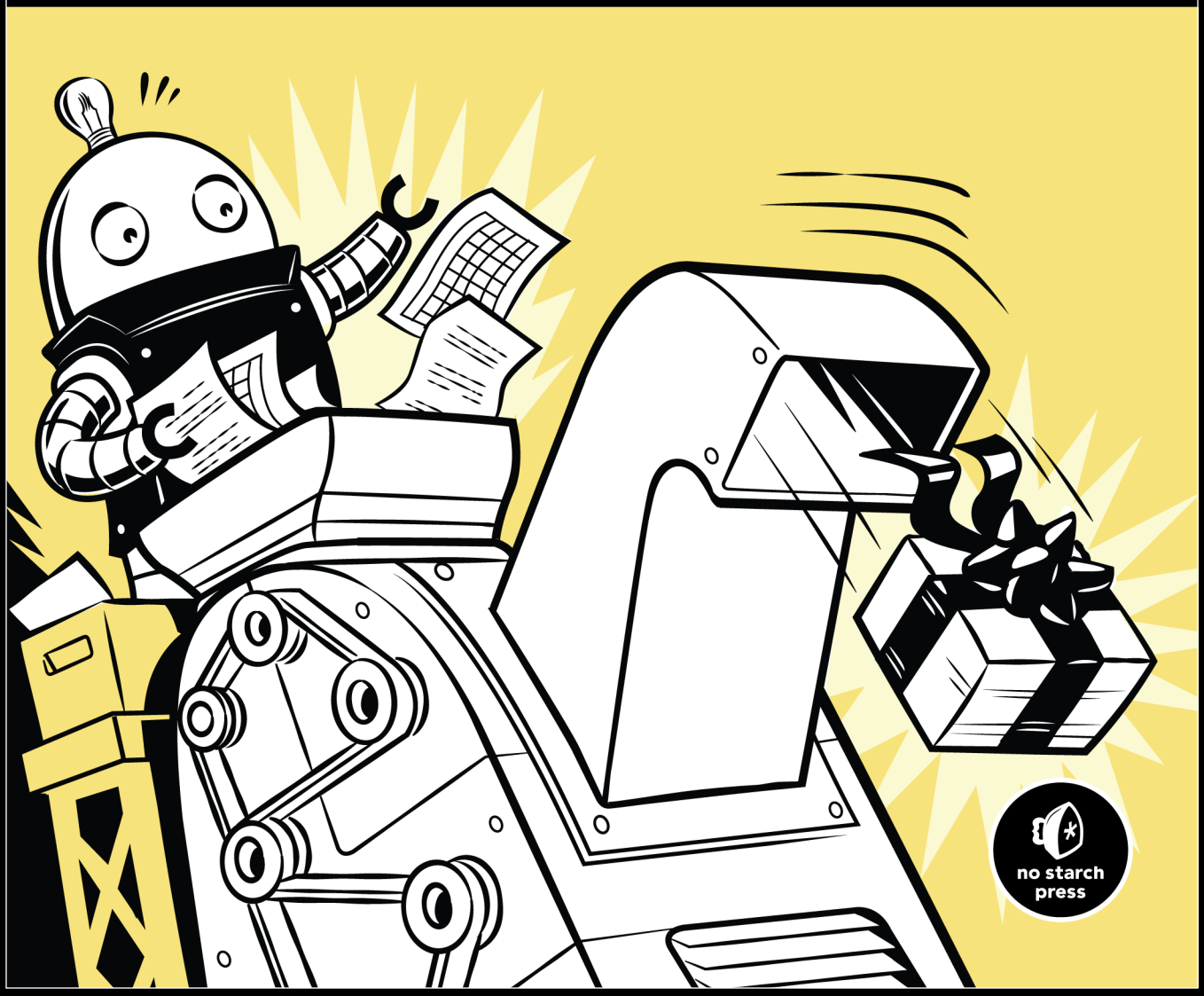
e.

```
print('hello'.find('oo'))
```

# PRACTICAL SQL

## A BEGINNER'S GUIDE TO *STORYTELLING* WITH *DATA*

ANTHONY DEBARROS



no starch press

# 1

# CREATING YOUR FIRST DATABASE AND TABLE

SQL is more than just a means for extracting knowledge from data. It's also a language for *defining* the structures that hold data so we can organize *relationships* in the data. Chief among those structures is the table.

A table is a grid of rows and columns that store data. Each row holds a collection of columns, and each column contains data of a specified type: most commonly, numbers, characters, and dates. We use SQL to define the structure of a table and how each table might relate to other tables in the database. We also use SQL to extract, or *query*, data from tables.

Understanding tables is fundamental to understanding the data in your database. Whenever I start working with a fresh database, the first thing I do is look at the tables within. I look for clues in the table names and their column structure. Do the tables contain text, numbers, or both? How many rows are in each table?

Next, I look at how many tables are in the database. The simplest database might have a single table. A full-bore application that handles

customer data or tracks air travel might have dozens or hundreds. The number of tables tells me not only how much data I'll need to analyze, but also hints that I should explore relationships among the data in each table.

Before you dig into SQL, let's look at an example of what the contents of tables might look like. We'll use a hypothetical database for managing a school's class enrollment; within that database are several tables that track students and their classes. The first table, called `student_enrollment`, shows the students that are signed up for each class section:

```
student_id    class_id      class_section    semester
----------    ----------    -------------    ---------
CHRISPA004    COMPSCI101    3                Fall 2017
DAVISHE010    COMPSCI101    3                Fall 2017
ABRILDA002    ENG101        40               Fall 2017
DAVISHE010    ENG101        40               Fall 2017
RILEYPH002    ENG101        40               Fall 2017
```

This table shows that two students have signed up for `COMPSCI101`, and three have signed up for `ENG101`. But where are the details about each student and class? In this example, these details are stored in separate tables called `students` and `classes`, and each table relates to this one. This is where the power of a *relational database* begins to show itself.

The first several rows of the `students` table include the following:

```
student_id    first_name    last_name    dob
----------    ----------    ---------    ----------
ABRILDA002    Abril         Davis        1999-01-10
CHRISPA004    Chris         Park         1996-04-10
DAVISHE010    Davis         Hernandez    1987-09-14
RILEYPH002    Riley         Phelps       1996-06-15
```

The `students` table contains details on each student, using the value in the `student_id` column to identify each one. That value acts as a unique *key* that connects both tables, giving you the ability to create rows such as the following with the `class_id` column from `student_enrollment` and the `first_name` and `last_name` columns from `students`:

```
class_id      first_name    last_name
----------    ----------    ---------
COMPSCI101    Davis         Hernandez
COMPSCI101    Chris         Park
ENG101        Abril         Davis
ENG101        Davis         Hernandez
ENG101        Riley         Phelps
```

The `classes` table would work the same way, with a `class_id` column and several columns of detail about the class. Database builders prefer to organize data using separate tables for each main *entity* the database manages in order to reduce redundant data. In the example, we store each student's name and date of birth just once. Even if the student signs up for multiple

classes—as Davis Hernandez did—we don't waste database space entering his name next to each class in the `student_enrollment` table. We just include his student ID.

Given that tables are a core building block of every database, in this chapter you'll start your SQL coding adventure by creating a table inside a new database. Then you'll load data into the table and view the completed table.

## Creating a Database

The PostgreSQL program you downloaded in the Introduction is a *database management system*, a software package that allows you to define, manage, and query databases. When you installed PostgreSQL, it created a *database server*—an instance of the application running on your computer—that includes a default database called `postgres`. The database is a collection of objects that includes tables, functions, user roles, and much more. According to the PostgreSQL documentation, the default database is "meant for use by users, utilities and third party applications" (see *https://www.postgresql.org/docs/current/static/app-initdb.html*). In the exercises in this chapter, we'll leave the default as is and instead create a new one. We'll do this to keep objects related to a particular topic or application organized together.

To create a database, you use just one line of SQL, shown in Listing 1-1. This code, along with all the examples in this book, is available for download via the resources at *https://www.nostarch.com/practicalSQL/*.

```
CREATE DATABASE analysis;
```

*Listing 1-1: Creating a database named `analysis`*

This statement creates a database on your server named `analysis` using default PostgreSQL settings. Note that the code consists of two keywords—`CREATE` and `DATABASE`—followed by the name of the new database. The statement ends with a semicolon, which signals the end of the command. The semicolon ends all PostgreSQL statements and is part of the ANSI SQL standard. Sometimes you can omit the semicolon, but not always, and particularly not when running multiple statements in the admin. So, using the semicolon is a good habit to form.

### Executing SQL in pgAdmin

As part of the Introduction to this book, you also installed the graphical administrative tool pgAdmin (if you didn't, go ahead and do that now). For much of our work, you'll use pgAdmin to run (or execute) the SQL statements we write. Later in the book in Chapter 16, I'll show you how to run SQL statements in a terminal window using the PostgreSQL command line program `psql`, but getting started is a bit easier with a graphical interface.

We'll use pgAdmin to run the SQL statement in Listing 1-1 that creates the database. Then, we'll connect to the new database and create a table. Follow these steps:

1. Run PostgreSQL. If you're using Windows, the installer set PostgreSQL to launch every time you boot up. On macOS, you must double-click *Postgres.app* in your Applications folder.
2. Launch pgAdmin. As you did in the Introduction, in the left vertical pane (the object browser) expand the plus sign to the left of the Servers node to show the default server. Depending on how you installed PostgreSQL, the default server may be named *localhost* or *PostgreSQL x*, where *x* is the version of the application.
3. Double-click the server name. If you supplied a password during installation, enter it at the prompt. You'll see a brief message that pgAdmin is establishing a connection.
4. In pgAdmin's object browser, expand **Databases** and click once on the `postgres` database to highlight it, as shown in Figure 1-1.
5. Open the Query Tool by choosing **Tools ▸ Query Tool**.
6. In the SQL Editor pane (the top horizontal pane), type or copy the code from Listing 1-1.
7. Click the lightning bolt icon to execute the statement. PostgreSQL creates the database, and in the Output pane in the Query Tool under Messages you'll see a notice indicating the query returned successfully, as shown in Figure 1-2.

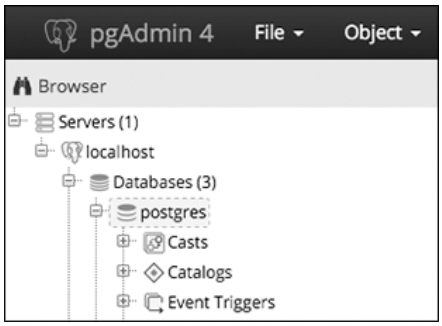



*Figure 1-1: Connecting to the default `postgres` database*

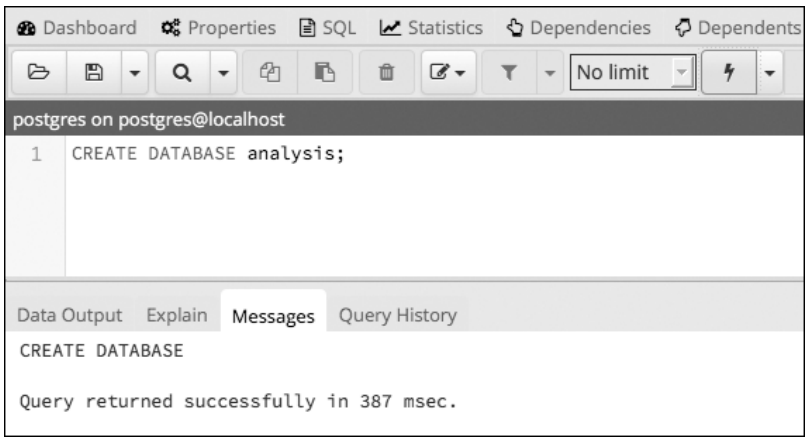



*Figure 1-2: Creating the `analysis` database*

8. To see your new database, right-click **Databases** in the object browser. From the pop-up menu, select **Refresh**, and the `analysis` database will appear in the list, as shown in Figure 1-3.

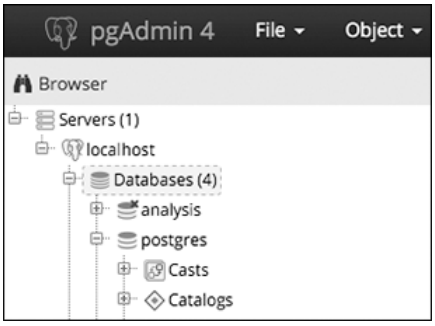



*Figure 1-3: The `analysis` database displayed in the object browser*

Good work! You now have a database called `analysis`, which you can use for the majority of the exercises in this book. In your own work, it's generally a best practice to create a new database for each project to keep tables with related data together.

### Connecting to the Analysis Database

Before you create a table, you must ensure that pgAdmin is connected to the `analysis` database rather than to the default `postgres` database.

To do that, follow these steps:

1. Close the Query Tool by clicking the **X** at the top right of the tool. You don't need to save the file when prompted.
2. In the object browser, click once on the `analysis` database.
3. Reopen the Query Tool by choosing **Tools ▸ Query Tool**.
4. You should now see the label `analysis on postgres@localhost` at the top of the Query Tool window. (Again, instead of `localhost`, your version may show `PostgreSQL`.)

Now, any code you execute will apply to the `analysis` database.

## Creating a Table

As I mentioned earlier, tables are where data lives and its relationships are defined. When you create a table, you assign a name to each *column* (sometimes referred to as a *field* or *attribute*) and assign it a *data type*. These are the values the column will accept—such as text, integers, decimals, and dates—and the definition of the data type is one way SQL enforces the integrity of data. For example, a column defined as `date` will take data in one of several standard formats, such as `YYYY-MM-DD`. If you try to enter characters not in a date format, for instance, the word `peach`, you'll receive an error.

Data stored in a table can be accessed and analyzed, or queried, with SQL statements. You can sort, edit, and view the data, and easily alter the table later if your needs change.

Let's make a table in the `analysis` database.

### The CREATE TABLE Statement

For this exercise, we'll use an often-discussed piece of data: teacher salaries. Listing 1-2 shows the SQL statement to create a table called `teachers`:

```
❶ CREATE TABLE teachers (
    ❷ id bigserial,
    ❸ first_name varchar(25),
      last_name varchar(50),
      school varchar(50),
    ❹ hire_date date,
    ❺ salary numeric
❻ );
```

*Listing 1-2: Creating a table named `teachers` with six columns*

This table definition is far from comprehensive. For example, it's missing several *constraints* that would ensure that columns that must be filled do indeed have data or that we're not inadvertently entering duplicate values. I cover constraints in detail in Chapter 7, but in these early chapters I'm omitting them to focus on getting you started on exploring data.

The code begins with the two SQL keywords ❶ `CREATE` and `TABLE` that, together with the name `teachers`, signal PostgreSQL that the next bit of code describes a table to add to the database. Following an opening parenthesis, the statement includes a comma-separated list of column names along with their data types. For style purposes, each new line of code is on its own line and indented four spaces, which isn't required, but it makes the code more readable.

Each column name represents one discrete data element defined by a data type. The `id` column ❷ is of data type `bigserial`, a special integer type that auto-increments every time you add a row to the table. The first row receives the value of `1` in the `id` column, the second row `2`, and so on. The `bigserial` data type and other serial types are PostgreSQL-specific implementations, but most database systems have a similar feature.

Next, we create columns for the teacher's first and last name, and the school where they teach ❸. Each is of the data type `varchar`, a text column with a maximum length specified by the number in parentheses. We're assuming that no one in the database will have a last name of more than 50 characters. Although this is a safe assumption, you'll discover over time that exceptions will always surprise you.

The teacher's `hire_date` ❹ is set to the data type `date`, and the `salary` column ❺ is a `numeric`. I'll cover data types more thoroughly in Chapter 3, but this table shows some common examples of data types. The code block wraps up ❻ with a closing parenthesis and a semicolon.

Now that you have a sense of how SQL looks, let's run this code in pgAdmin.

### Making the teachers Table

You have your code and you're connected to the database, so you can make the table using the same steps we did when we created the database:

1. Open the pgAdmin Query Tool (if it's not open, click once on the `analysis` database in pgAdmin's object browser, and then choose **Tools ▸ Query Tool**).
2. Copy the `CREATE TABLE` script from Listing 1-2 into the SQL Editor.
3. Execute the script by clicking the lightning bolt icon.

If all goes well, you'll see a message in the pgAdmin Query Tool's bottom output pane that reads, `Query returned successfully with no result in 84 msec`. Of course, the number of milliseconds will vary depending on your system.

Now, find the table you created. Go back to the main pgAdmin window and, in the object browser, right-click the `analysis` database and choose **Refresh**. Choose **Schemas ▸ public ▸ Tables** to see your new table, as shown in Figure 1-4.

Expand the `teachers` table node by clicking the plus sign to the left of its name. This reveals more details about the table, including the column names, as shown in Figure 1-5. Other information appears as well, such as indexes, triggers, and constraints, but I'll cover those in later chapters. Clicking on the table name and then selecting the **SQL** menu in the pgAdmin workspace will display the SQL statement used to make the `teachers` table.

Congratulations! So far, you've built a database and added a table to it. The next step is to add data to the table so you can write your first query.

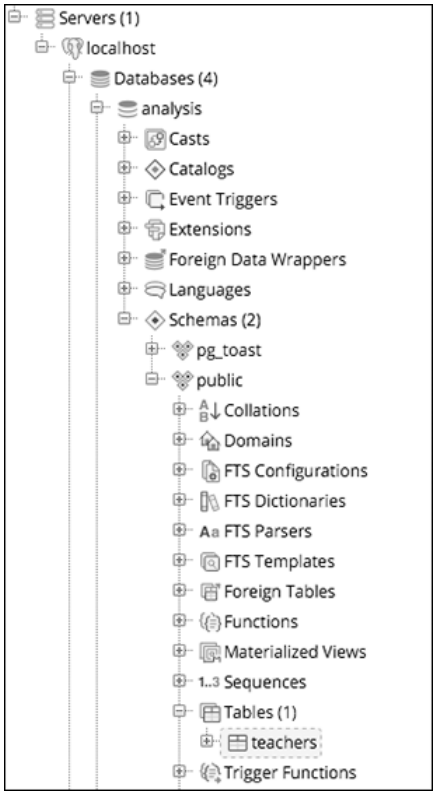



*Figure 1-4: The `teachers` table in the object browser*

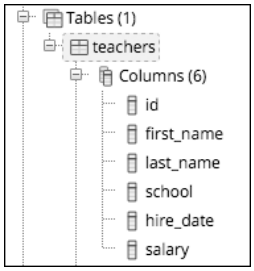



*Figure 1-5: Table details for `teachers`*

## Inserting Rows into a Table

You can add data to a PostgreSQL table in several ways. Often, you'll work with a large number of rows, so the easiest method is to import data from a text file or another database directly into a table. But just to get started, we'll add a few rows using an `INSERT INTO ... VALUES` statement that specifies the target columns and the data values. Then we'll view the data in its new home.

### *The INSERT Statement*

To insert some data into the table, you first need to erase the `CREATE TABLE` statement you just ran. Then, following the same steps as you did to create the database and table, copy the code in Listing 1-3 into your pgAdmin Query Tool:

```
❶ INSERT INTO teachers (first_name, last_name, school, hire_date, salary)
❷ VALUES ('Janet', 'Smith', 'F.D. Roosevelt HS', '2011-10-30', 36200),
       ('Lee', 'Reynolds', 'F.D. Roosevelt HS', '1993-05-22', 65000),
       ('Samuel', 'Cole', 'Myers Middle School', '2005-08-01', 43500),
       ('Samantha', 'Bush', 'Myers Middle School', '2011-10-30', 36200),
       ('Betty', 'Diaz', 'Myers Middle School', '2005-08-30', 43500),
       ('Kathleen', 'Roush', 'F.D. Roosevelt HS', '2010-10-22', 38500);❸
```

*Listing 1-3: Inserting data into the teachers table*

This code block inserts names and data for six teachers. Here, the PostgreSQL syntax follows the ANSI SQL standard: after the `INSERT INTO` keywords is the name of the table, and in parentheses are the columns to be filled ❶. In the next row is the `VALUES` keyword and the data to insert into each column in each row ❷. You need to enclose the data for each row in a set of parentheses, and inside each set of parentheses, use a comma to separate each column value. The order of the values must also match the order of the columns specified after the table name. Each row of data ends with a comma, and the last row ends the entire statement with a semicolon ❸.

Notice that certain values that we're inserting are enclosed in single quotes, but some are not. This is a standard SQL requirement. Text and dates require quotes; numbers, including integers and decimals, don't require quotes. I'll highlight this requirement as it comes up in examples. Also, note the date format we're using: a four-digit year is followed by the month and date, and each part is joined by a hyphen. This is the international standard for date formats; using it will help you avoid confusion. (Why is it best to use the format *YYYY-MM-DD*? Check out *https://xkcd.com/1179/* to see a great comic about it.) PostgreSQL supports many additional date formats, and I'll use several in examples.

You might be wondering about the `id` column, which is the first column in the table. When you created the table, your script specified that column to be the `bigserial` data type. So as PostgreSQL inserts each row, it automatically fills the `id` column with an auto-incrementing integer. I'll cover that in detail in Chapter 3 when I discuss data types.

Now, run the code. This time the message in the Query Tool should include the words `Query returned successfully: 6 rows affected`.

### Viewing the Data

You can take a quick look at the data you just loaded into the `teachers` table using pgAdmin. In the object browser, locate the table and right-click. In the pop-up menu, choose **View/Edit Data ▸ All Rows**. As Figure 1-6 shows, you'll see the six rows of data in the table with each column filled by the values in the SQL statement.

Data Output | Explain | Messages | Query History

| | id bigint | first_name character vary | last_name character vary | school character varying | hire_date date | salary numeric |
|---|---|---|---|---|---|---|
| 1 | 1 | Janet | Smith | F.D. Roosevelt ... | 2011-10-30 | 36200 |
| 2 | 2 | Lee | Reynolds | F.D. Roosevelt ... | 1993-05-22 | 65000 |
| 3 | 3 | Samuel | Cole | Myers Middle S... | 2005-08-01 | 43500 |
| 4 | 4 | Samantha | Bush | Myers Middle S... | 2011-10-30 | 36200 |
| 5 | 5 | Betty | Diaz | Myers Middle S... | 2005-08-30 | 43500 |
| 6 | 6 | Kathleen | Roush | F.D. Roosevelt ... | 2010-10-22 | 38500 |

*Figure 1-6: Viewing table data directly in pgAdmin*

Notice that even though you didn't insert a value for the `id` column, each teacher has an ID number assigned.

You can view data using the pgAdmin interface in a few ways, but we'll focus on writing SQL to handle those tasks.

## When Code Goes Bad

There may be a universe where code always works, but unfortunately, we haven't invented a machine capable of transporting us there. Errors happen. Whether you make a typo or mix up the order of operations, computer languages are unforgiving about syntax. For example, if you forget a comma in the code in Listing 1-3, PostgreSQL squawks back an error:

```
ERROR:  syntax error at or near "("
LINE 5:     ('Samuel', 'Cole', 'Myers Middle School', '2005-08-01', 43...
            ^
********** Error **********
```

Fortunately, the error message hints at what's wrong and where: a syntax error is near an open parenthesis on line 5. But sometimes error messages can be more obscure. In that case, you do what the best coders do: a quick internet search for the error message. Most likely, someone else has experienced the same issue and might know the answer.

## Formatting SQL for Readability

SQL requires no special formatting to run, so you're free to use your own psychedelic style of uppercase, lowercase, and random indentations. But that won't win you any friends when others need to work with your code (and sooner or later someone will). For the sake of readability and being a good coder, it's best to follow these conventions:

- Uppercase SQL keywords, such as `SELECT`. Some SQL coders also uppercase the names of data types, such as `TEXT` and `INTEGER`. I use lowercase characters for data types in this book to separate them in your mind from keywords, but you can uppercase them if desired.
- Avoid camel case and instead use `lowercase_and_underscores` for object names, such as tables and column names (see more details about case in Chapter 7).
- Indent clauses and code blocks for readability using either two or four spaces. Some coders prefer tabs to spaces; use whichever works best for you or your organization.

We'll explore other SQL coding conventions as we go through the book, but these are the basics.

## Wrapping Up

You accomplished quite a bit in this first chapter: you created a database and a table, and then loaded data into it. You're on your way to adding SQL to your data analysis toolkit! In the next chapter, you'll use this set of teacher data to learn the basics of querying a table using `SELECT`.

**TRY IT YOURSELF**

Here are two exercises to help you explore concepts related to databases, tables, and data relationships:

1. Imagine you're building a database to catalog all the animals at your local zoo. You want one table to track the kinds of animals in the collection and another table to track the specifics on each animal. Write `CREATE TABLE` statements for each table that include some of the columns you need. Why did you include the columns you chose?
2. Now create `INSERT` statements to load sample data into the tables. How can you view the data via the pgAdmin tool? Create an additional `INSERT` statement for one of your tables. Purposely omit one of the required commas separating the entries in the `VALUES` clause of the query. What is the error message? Would it help you find the error in the code?

# Serious Cryptography

## A Practical Introduction to Modern Encryption

Jean-Philippe Aumasson

Foreword by Matthew D. Green

no starch press

# 4

# BLOCK CIPHERS

During the Cold War, the US and Soviets developed their own ciphers. The US government created the Data Encryption Standard (DES), which was adopted as a federal standard from 1979 to 2005, while the KGB developed GOST 28147-89, an algorithm kept secret until 1990 and still used today. In 2000, the US-based National Institute of Standards and Technology (NIST) selected the successor to DES, called the *Advanced Encryption Standard (AES)*, an algorithm developed in Belgium and now found in most electronic devices. AES, DES, and GOST 28147-89 have something in common: they're all *block ciphers*, a type of cipher that combines a core algorithm working on blocks of data with a mode of operation, or a technique to process sequences of data blocks.

This chapter reviews the core algorithms that underlie block ciphers, discusses their modes of operation, and explains how they all work together. It also discusses how AES works and concludes with coverage of a classic attack tool from the 1970s, the meet-in-the-middle attack, and a favorite attack technique of the 2000s—padding oracles.

## What Is a Block Cipher?

A block cipher consists of an encryption algorithm and a decryption algorithm:

- The *encryption algorithm* (**E**) takes a key, *K*, and a plaintext block, *P*, and produces a ciphertext block, *C*. We write an encryption operation as $C = \mathbf{E}(K, P)$.
- The *decryption algorithm* (**D**) is the inverse of the encryption algorithm and decrypts a message to the original plaintext, *P*. This operation is written as $P = \mathbf{D}(K, C)$.

Since they're the inverse of each other, the encryption and decryption algorithms usually involve similar operations.

### Security Goals

If you've followed earlier discussions about encryption, randomness, and indistinguishability, the definition of a secure block cipher will come as no surprise. Again, we'll define security as random-lookingness, so to speak.

In order for a block cipher to be secure, it should be a *pseudorandom permutation (PRP)*, meaning that as long as the key is secret, an attacker shouldn't be able to compute an output of the block cipher from any input. That is, as long as *K* is secret and random from an attacker's perspective, they should have no clue about what $\mathbf{E}(K, P)$ looks like, for any given *P*.

More generally, attackers should be unable to discover any *pattern* in the input/output values of a block cipher. In other words, it should be impossible to tell a block cipher from a truly random permutation, given black-box access to the encryption and decryption functions for some fixed and unknown key. By the same token, they should be unable to recover a secure block cipher's secret key; otherwise, they would be able to use that key to tell the block cipher from a random permutation. Of course that also implies that attackers can't predict the plaintext that corresponds to a given ciphertext produced by the block cipher.

### Block Size

Two values characterize a block cipher: the block size and the key size. Security depends on both values. Most block ciphers have either 64-bit or 128-bit blocks—DES's blocks have 64 ($2^6$) bits, and AES's blocks have 128 ($2^7$) bits. In computing, lengths that are powers of two simplify data processing, storage, and addressing. But why $2^6$ and $2^7$ and not $2^4$ or $2^{16}$ bits?

For one thing, it's important that blocks are not too large in order to minimize both the length of ciphertext and the memory footprint. With regard to the length of the ciphertext, block ciphers process blocks, not bits. This means that in order to encrypt a 16-bit message when blocks are 128 bits, you'll first need to convert the message into a 128-bit block, and only then will the block cipher process it and return a 128-bit ciphertext. The wider the blocks, the longer this overhead. As for the *memory footprint*, in order to process a 128-bit block, you need at least 128 bits of memory. This is small enough to fit in the registers of most CPUs or to be implemented using dedicated hardware circuits. Blocks of 64, 128, or even 512 bits are short enough to allow for efficient implementations in most cases. But larger blocks (for example, several kilobytes long) can have a noticeable impact on the cost and performance of implementations.

When ciphertexts' length or memory footprint is critical, you may have to use 64-bit blocks, because these will produce shorter ciphertexts and consume less memory. Otherwise, 128-bit or larger blocks are better, mainly because 128-bit blocks can be processed more efficiently than 64-bit ones on modern CPUs and are also more secure. In particular, CPUs can leverage special CPU instructions in order to efficiently process one or more 128-bit blocks in parallel—for example, the Advanced Vector Extensions (AVX) family of instructions in Intel CPUs.

### The Codebook Attack

While blocks shouldn't be too large, they also shouldn't be too small; otherwise, they may be susceptible to *codebook attacks*, which are attacks against block ciphers that are only efficient when smaller blocks are used. The codebook attack works like this with 16-bit blocks:

1. Get the 65536 ($2^{16}$) ciphertexts corresponding to each 16-bit plaintext block.
2. Build a lookup table—the *codebook*—mapping each ciphertext block to its corresponding plaintext block.
3. To decrypt an unknown ciphertext block, look up its corresponding plaintext block in the table.

When 16-bit blocks are used, the lookup table needs only $2^{16} \times 16 = 2^{20}$ bits of memory, or 128 kilobytes. With 32-bit blocks, memory needs grow to 16 gigabytes, which is still manageable. But with 64-bit blocks, you'd have to store $2^{70}$ bits (a zetabit, or 128 exabytes), so forget about it. Codebook attacks won't be an issue for larger blocks.

## How to Construct Block Ciphers

There are hundreds of block ciphers but only a handful of techniques to construct one. First, a block cipher used in practice isn't a gigantic algorithm but a repetition of *rounds*, a short sequence of operations that is weak on its

own but strong in number. Second, there are two main techniques to construct a round: substitution–permutation networks (as in AES) and Feistel schemes (as in DES). In this section, we look at how these work, after viewing an attack that works when all rounds are identical to each other.

### A Block Cipher's Rounds

Computing a block cipher boils down to computing a sequence of *rounds*. In a block cipher, a round is a basic transformation that is simple to specify and to implement, and which is iterated several times to form the block cipher's algorithm. This construction, consisting of a small component repeated many times, is simpler to implement and to analyze than a construction that would consist of a single huge algorithm.

For example, a block cipher with three rounds encrypts a plaintext by computing $C = \mathbf{R}_3(\mathbf{R}_2(\mathbf{R}_1(P)))$, where the rounds are $\mathbf{R}_1$, $\mathbf{R}_2$, and $\mathbf{R}_3$ and $P$ is a plaintext. Each round should also have an inverse in order to make it possible for a recipient to compute back to plaintext. Specifically, $P = \mathbf{iR}_1(\mathbf{iR}_2(\mathbf{iR}_3(C)))$, where $\mathbf{iR}_1$ is the inverse of $\mathbf{R}_1$, and so on.

The round functions—$\mathbf{R}_1$, $\mathbf{R}_2$, and so on—are usually identical algorithms, but they are parameterized by a value called the *round key*. Two round functions with two distinct round keys will behave differently, and therefore will produce distinct outputs if fed with the same input.

Round keys are keys derived from the main key, $K$, using an algorithm called a *key schedule*. For example, $\mathbf{R}_1$ takes the round key $K_1$, $\mathbf{R}_2$ takes the round key $K_2$, and so on.

Round keys should always be different from each other in every round. For that matter, not all round keys should be equal to the key $K$. Otherwise, all the rounds would be identical and the block cipher would be less secure, as described next.

### The Slide Attack and Round Keys

In a block cipher, no round should be identical to another round in order to avoid a *slide attack*. Slide attacks look for two plaintext/ciphertext pairs $(P_1, C_1)$ and $(P_2, C_2)$, where $P_2 = \mathbf{R}(P_1)$ if $\mathbf{R}$ is the cipher's round (see Figure 4-1). When rounds are identical, the relation between the two plaintexts, $P_2 = \mathbf{R}(P_1)$, implies the relation $C_2 = \mathbf{R}(C_1)$ between their respective ciphertexts. Figure 4-1 shows three rounds, but the relation $C_2 = \mathbf{R}(C_1)$ will hold no matter the number of rounds, be it 3, 10, or 100. The problem is that knowing the input and output of a single round often helps recover the key. (For details, read the 1999 paper by Biryukov and Wagner called "Advanced Slide Attacks," available at *https://www.iacr.org/archive/eurocrypt2000/1807/18070595-new.pdf*)

The use of different round keys as parameters ensures that the rounds will behave differently and thus foil slide attacks.

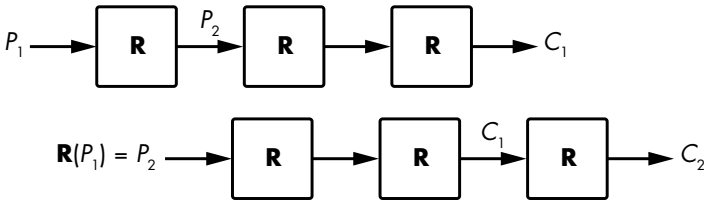



*Figure 4-1: The principle of the slide attack, against block ciphers with identical rounds*

**NOTE** *One potential byproduct and benefit of using round keys is protection against side-channel attacks, or attacks that exploit information leaked from the implementation of a cipher (for example, electromagnetic emanations). If the transformation from the main key,* K*, to a round key,* $K_i$*, is not invertible, then if an attacker finds* $K_i$*, they can't use that key to find* K*. Unfortunately, few block ciphers have a one-way key schedule. The key schedule of AES allows attackers to compute* K *from any round key,* $K_i$*, for example.*

### Substitution–Permutation Networks

If you've read textbooks about cryptography, you'll undoubtedly have read about *confusion* and *diffusion*. Confusion means that the input (plaintext and encryption key) undergoes complex transformations, and diffusion means that these transformations depend equally on all bits of the input. At a high level, confusion is about depth whereas diffusion is about breadth. In the design of a block cipher, confusion and diffusion take the form of substitution and permutation operations, which are combined within substitution–permutation networks (SPNs).

Substitution often appears in the form of *S-boxes*, or *substitution boxes*, which are small lookup tables that transform chunks of 4 or 8 bits. For example, the first of the eight S-boxes of the block cipher Serpent is composed of the 16 elements (3 8 f 1 a 6 5 b e d 4 2 7 0 9 c), where each element represents a 4-bit nibble. This particular S-box maps the 4-bit nibble 0000 to 3 (0011), the 4-bit nibble 0101 (5 in decimal) to 6 (0110), and so on.

**NOTE** *S-boxes must be carefully chosen to be cryptographically strong: they should be as nonlinear as possible (inputs and outputs should be related with complex equations) and have no statistical bias (meaning, for example, that flipping an input bit should potentially affect any of the output bits).*

The permutation in a substitution–permutation network can be as simple as changing the order of the bits, which is easy to implement but doesn't mix up the bits very much. Instead of a reordering of the bits, some ciphers use basic linear algebra and matrix multiplications to mix up the bits: they perform a series of multiplication operations with fixed values (the matrix's

coefficients) and then add the results. Such linear algebra operations can quickly create dependencies between all the bits within a cipher and thus ensure strong diffusion. For example, the block cipher FOX transforms a 4-byte vector $(a, b, c, d)$ to $(a', b', c', d')$, defined as follows:

$$
\begin{aligned}
a' &= a + b + c + (2 \times d) \\
b' &= a + (253 \times b) + (2 \times c) + d \\
c' &= (253 \times a) + (2 \times b) + c + d \\
d' &= (2 \times a) + b + (253 \times c) + d
\end{aligned}
$$

In the above equations, the numbers 2 and 253 are interpreted as binary polynomials rather than integers; hence, additions and multiplications are defined a bit differently than what we're used to. For example, instead of having 2 + 2 = 4, we have 2 + 2 = 0. Regardless, the point is that each byte in the initial state affects all 4 bytes in the final state.

## Feistel Schemes

In the 1970s, IBM engineer Horst Feistel designed a block cipher called Lucifer that works as follows:

1. Split the 64-bit block into two 32-bit halves, $L$ and $R$.
2. Set $L$ to $L \oplus \mathbf{F}(R)$, where **F** is a substitution–permutation round.
3. Swap the values of $L$ and $R$.
4. Go to step 2 and repeat 15 times.
5. Merge $L$ and $R$ into the 64-bit output block.

This construction became known as a *Feistel scheme*, as shown in Figure 4-2. The left side is the scheme as just described; the right side is a functionally equivalent representation where, instead of swapping $L$ and $R$, rounds alternate the operations $L = L \oplus \mathbf{F}(R)$ and $R = R \oplus \mathbf{F}(L)$.

I've omitted the keys from Figure 4-2 to simplify the diagrams, but note that the first **F** takes a first round key, $K_1$, and the second **F** takes another round key, $K_2$. In DES, the **F** functions take a 48-bit round key, which is derived from the 56-bit key, $K$.

In a Feistel scheme, the **F** function can be either a pseudorandom permutation (PRP) or a pseudorandom function (PRF). A PRP yields distinct outputs for any two distinct inputs, whereas a PRF will have values $X$ and $Y$ for which $\mathbf{F}(X) = \mathbf{F}(Y)$. But in a Feistel scheme, that difference doesn't matter as long as **F** is cryptographically strong.

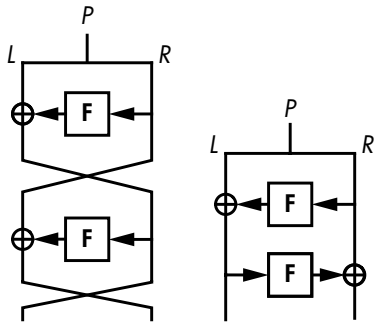



*Figure 4-2: The Feistel scheme block cipher construction in two equivalent forms*

How many rounds should there be in a Feistel scheme? Well, DES performs 16 rounds, whereas GOST 28147-89 performs 32 rounds. If the **F** function is as strong as possible, four rounds are in theory sufficient, but real ciphers use more rounds to defend against potential weaknesses in **F**.

## The Advanced Encryption Standard (AES)

AES is the most-used cipher in the universe. Prior to the adoption of AES, the standard cipher in use was DES, with its ridiculous 56-bit security, as well as the upgraded version of DES known as Triple DES, or 3DES.

Although 3DES provides a higher level of security (112-bit security), it's inefficient because the key needs to be 168 bits long in order to get 112-bit security, and it's slow in software (DES was created to be fast in integrated circuits, not on mainstream CPUs). AES fixes both issues.

NIST standardized AES in 2000 as a replacement for DES, at which point it became the world's de facto encryption standard. Most commercial encryption products today support AES, and the NSA has approved it for protecting top-secret information. (Some countries do prefer to use their own cipher, largely because they don't want to use a US standard, but AES is actually more Belgian than it is American.)

**NOTE** *AES used to be called Rijndael (a portmanteau for its inventors' names, Rijmen and Daemen, pronounced like "rain-dull") when it was one of the 15 candidates in the AES competition, the process held by NIST from 1997 to 2000 to specify "an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive government information well into the next century," as stated in the 1997 announcement of the competition in the Federal Register. The AES competition was kind of a "Got Talent" competition for cryptographers, where anyone could participate by submitting a cipher or breaking other contestants' ciphers.*

### AES Internals

AES processes blocks of 128 bits using a secret key of 128, 192, or 256 bits, with the 128-bit key being the most common because it makes encryption slightly faster and because the difference between 128- and 256-bit security is meaningless for most applications.

Whereas some ciphers work with individual bits or 64-bit words, AES manipulates *bytes*. It views a 16-byte plaintext as a two-dimensional array of bytes ($s = s_0, s_1, \ldots, s_{15}$), as shown in Figure 4-3. (The letter $s$ is used because this array is called the *internal state*, or just *state*.) AES transforms the bytes, columns, and rows of this array to produce a final value that is the ciphertext.

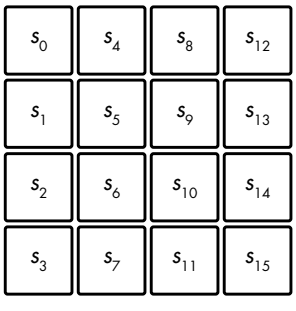



*Figure 4-3: The internal state of AES viewed as a 4 × 4 array of 16 bytes*

In order to transform its state, AES uses an SPN structure like the one shown in Figure 4-4, with 10 rounds for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys.

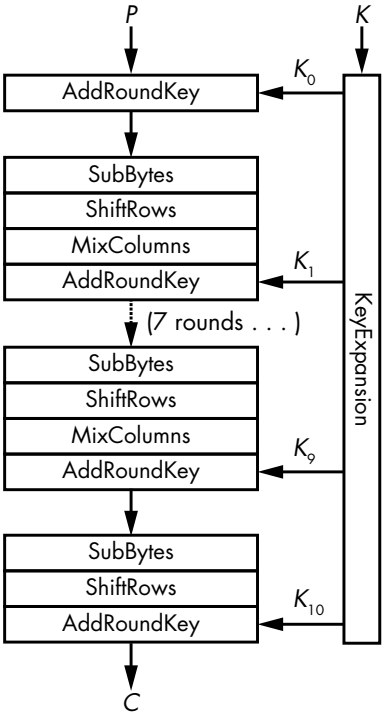



*Figure 4-4: The internal operations of AES*

Figure 4-4 shows the four building blocks of an AES round (note that all but the last round are a sequence of SubBytes, ShiftRows, MixColumns, and AddRoundKey):

**AddRoundKey** XORs a round key to the internal state.

**SubBytes** Replaces each byte ($s_0$, $s_1$, . . . , $s_{15}$) with another byte according to an S-box. In this example, the S-box is a lookup table of 256 elements.

**ShiftRows** Shifts the $i$th row of $i$ positions, for $i$ ranging from 0 to 3 (see Figure 4-5).

**MixColumns** Applies the same linear transformation to each of the four columns of the state (that is, each group of cells with the same shade of gray, as shown on the left side of Figure 4-5).

Remember that in an SPN, the *S* stands for substitution and the *P* for permutation. Here, the substitution layer is SubBytes and the permutation layer is the combination of ShiftRows and MixColumns.

The key schedule function *KeyExpansion*, shown in Figure 4-4, is the AES key schedule algorithm. This expansion creates 11 round keys ($K_0$, $K_1$, . . . , $K_{10}$) of 16 bytes each from the 16-byte key, using the same S-box as SubBytes and a combination of XORs. One important property of

KeyExpansion is that given any round key, $K_i$, an attacker can determine all other round keys as well as the main key, $K$, by reversing the algorithm. The ability to get the key from any round key is usually seen as an imperfect defense against side-channel attacks, where an attacker may easily recover a round key.

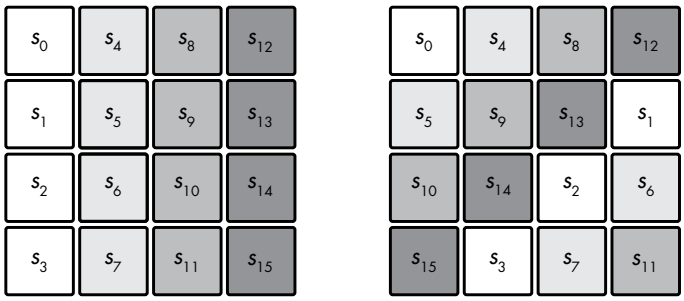



*Figure 4-5: ShiftRows rotates bytes within each row of the internal state.*

Without these operations, AES would be totally insecure. Each operation contributes to AES's security in a specific way:

- Without KeyExpansion, all rounds would use the same key, $K$, and AES would be vulnerable to slide attacks.
- Without AddRoundKey, encryption wouldn't depend on the key; hence, anyone could decrypt any ciphertext without the key.
- SubBytes brings nonlinear operations, which add cryptographic strength. Without it, AES would just be a large system of linear equations that is solvable using high-school algebra.
- Without ShiftRows, changes in a given column would never affect the other columns, meaning you could break AES by building four $2^{32}$-element codebooks for each column. (Remember that in a secure block cipher, flipping a bit in the input should affect all the output bits.)
- Without MixColumns, changes in a byte would not affect any other bytes of the state. A chosen-plaintext attacker could then decrypt any ciphertext after storing 16 lookup tables of 256 bytes each that hold the encrypted values of each possible value of a byte.

Notice in Figure 4-4 that the last round of AES doesn't include the MixColumns operation. That operation is omitted in order to save useless computation: because MixColumns is linear (meaning, predictable), you could cancel its effect in the very last round by combining bits in a way that doesn't depend on their value or the key. SubBytes, however, can't be inverted without the state's value being known prior to AddRoundKey.

To decrypt a ciphertext, AES unwinds each operation by taking its inverse function: the inverse lookup table of SubBytes reverses the SubBytes transformation, ShiftRow shifts in the opposite direction,

MixColumns's inverse is applied (as in the matrix inverse of the matrix encoding its operation), and AddRoundKey's XOR is unchanged because the inverse of an XOR is another XOR.

### AES in Action

To try encrypting and decrypting with AES, you can use Python's cryptography library, as in Listing 4-1.

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from binascii import hexlify as hexa
from os import urandom

# pick a random 16-byte key using Python's crypto PRNG
k = urandom(16)
print "k = %s" % hexa(k)
# create an instance of AES-128 to encrypt a single block
cipher = Cipher(algorithms.AES(k), modes.ECB(), backend = default_backend())
aes_encrypt = cipher.encryptor()

# set plaintext block p to the all-zero string
p = '\x00'*16
# encrypt plaintext p to ciphertext c
c = aes_encrypt.update(p) + aes_encrypt.finalize()
print "enc(%s) = %s" % (hexa(p), hexa(c))
# decrypt ciphertext c to plaintext p
aes_decrypt = cipher.decryptor()
p = aes_decrypt.update(c) + aes_decrypt.finalize()
print "dec(%s) = %s" % (hexa(c), hexa(p))
```

*Listing 4-1: Trying AES with Python's cryptography library*

Running this script produces something like the following output:

```
$ ./aes_block.py
k = 2c6202f9a582668aa96d511862d8a279
enc(00000000000000000000000000000000) = 12b620bb5eddcde9a07523e59292a6d7
dec(12b620bb5eddcde9a07523e59292a6d7) = 00000000000000000000000000000000
```

You'll get different results because the key is randomized at every new execution.

## Implementing AES

Real AES software works differently than the algorithm shown in Figure 4-4. You won't find production-level AES code calling a `SubBytes()` function, then a `ShiftRows()` function, and then a `MixColumns()` function because that would be inefficient. Instead, fast AES software uses special techniques called table-based implementations and native instructions.

### Table-Based Implementations

Table-based implementations of AES replace the sequence SubBytes-ShiftRows-MixColumns with a combination of XORs and lookups in tables hardcoded into the program and loaded in memory at execution time. This is possible because MixColumns is equivalent to XORing four 32-bit values, where each depends on a single byte from the state and on SubBytes. Thus, you can build four tables with 256 entries each, one for each byte value, and implement the sequence SubBytes-MixColumns by looking up four 32-bit values and XORing them together.

For example, the table-based C implementation in the OpenSSL toolkit looks like Listing 4-2.

```
    /* round 1: */
    t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
    t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
    t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
    t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
    /* round 2: */
    s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[ 8];
    s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^ Te3[t0 & 0xff] ^ rk[ 9];
    s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^ Te3[t1 & 0xff] ^ rk[10];
    s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^ Te3[t2 & 0xff] ^ rk[11];
--snip--
```

*Listing 4-2: The table-based C implementation of AES in OpenSSL*

A basic table-based implementation of AES encryption needs four kilobytes' worth of tables because each table stores 256 32-bit values, which occupy 256 × 32 = 8192 bits, or one kilobyte. Decryption requires another four tables, and thus four more kilobytes. But there are tricks to reduce the storage from four kilobytes to one, or even fewer.

Alas, table-based implementations are vulnerable to *cache-timing attacks*, which exploit timing variations when a program reads or writes elements in cache memory. Depending on the relative position in cache memory of the elements accessed, access time varies. Timings thus leak information about which element was accessed, which in turn leaks information on the secrets involved.

Cache-timing attacks are difficult to avoid. One obvious solution would be to ditch lookup tables altogether by writing a program whose execution time doesn't depend on its inputs, but that's almost impossible to do and still retain the same speed, so chip manufacturers have opted for a radical solution: instead of relying on potentially vulnerable software, they rely on *hardware*.

### Native Instructions

AES native instructions (AES-NI) solve the problem of cache-timing attacks on AES software implementations. To understand how AES-NI works, you need to think about the way software runs on hardware: to run a program, a

microprocessor translates binary code into a series of instructions executed by integrated circuit components. For example, a `MUL` assembly instruction between two 32-bit values will activate the transistors implementing a 32-bit multiplier in the microprocessor. To implement a crypto algorithm, we usually just express a combination of such basic operations—additions, multiplications, XORs, and so on—and the microprocessor activates its adders, multipliers, and XOR circuit in the prescribed order.

AES native instructions take this to a whole new level by providing developers with dedicated assembly instructions that compute AES. Instead of coding an AES round as a sequence of assembly instructions, when using AES-NI, you just call the instruction `AESENC` and the chip will compute the round for you. Native instructions allow you to just tell the processor to run an AES round instead of requiring you to program rounds as a combination of basic operations.

A typical assembly implementation of AES using native instructions looks like Listing 4-3.

```
PXOR       %xmm5,  %xmm0
AESENC     %xmm6,  %xmm0
AESENC     %xmm7,  %xmm0
AESENC     %xmm8,  %xmm0
AESENC     %xmm9,  %xmm0
AESENC     %xmm10, %xmm0
AESENC     %xmm11, %xmm0
AESENC     %xmm12, %xmm0
AESENC     %xmm13, %xmm0
AESENC     %xmm14, %xmm0
AESENCLAST %xmm15, %xmm0
```

*Listing 4-3: AES native instructions*

This code encrypts the 128-bit plaintext initially in the register `xmm0`, assuming that registers `xmm5` to `xmm15` hold the precomputed round keys, with each instruction writing its result into `xmm0`. The initial `PXOR` instruction XORs the first round key prior to computing the first round, and the final `AESENCLAST` instruction performs the last round slightly different from the others (MixColumns is omitted).

**NOTE** *AES is about ten times faster on platforms that implement native instructions, which as I write this, are virtually all laptop, desktop, and server microprocessors, as well as most mobile phones and tablets. In fact, on the latest Intel microarchitecture the `AESENC` instruction has a latency of four cycles with a reciprocal throughput of one cycle, meaning that a call to `AESENC` takes four cycles to complete and that a new call can be made every cycle. To encrypt a series of blocks consecutively it thus takes 4 × 10 = 40 cycles to complete the 10 rounds or 40 / 16 = 2.5 cycles per byte. At 2 GHz ($2 \times 10^9$ cycles per second), that gives a throughput of about 736 megabytes per second. If the blocks to encrypt or decrypt are independent of each other, as certain modes of operation allow, then four blocks can be processed in parallel to take full advantage of the `AESENC` circuit in order to reach a latency of 10 cycles per block instead of 40, or about 3 gigabytes per second.*

### Is AES Secure?

AES is as secure as a block cipher can be, and it will never be broken. Fundamentally, AES is secure because all output bits depend on all input bits in some complex, pseudorandom way. To achieve this, the designers of AES carefully chose each component for a particular reason—MixColumns for its maximal diffusion properties and SubBytes for its optimal non-linearity—and they have shown that this composition protects AES against whole classes of cryptanalytic attacks.

But there's no proof that AES is immune to all possible attacks. For one thing, we don't know what all possible attacks are, and we don't always know how to prove that a cipher is secure against a given attack. The only way to really gain confidence in the security of AES is to crowdsource attacks: have many skilled people attempt to break AES and, hopefully, fail to do so.

After more than 15 years and hundreds of research publications, the theoretical security of AES has only been scratched. In 2011 cryptanalysts found a way to recover an AES-128 key by performing about $2^{126}$ operations instead of $2^{128}$, a speed-up of a factor four. But this "attack" requires an insane amount of plaintext–ciphertext pairs—about $2^{88}$ bits worth. In other words, it's a nice finding but not one you need to worry about.

The upshot is that you should care about a million things when implementing and deploying crypto, but AES security is not one of those. The biggest threat to block ciphers isn't in their core algorithms but in their modes of operation. When an incorrect mode is chosen, or when the right one is misused, even a strong cipher like AES won't save you.

## Modes of Operation

In Chapter 1, I explained how encryption schemes combine a permutation with a mode of operation to handle messages of any length. In this section, I'll cover the main modes of operations used by block ciphers, their security and function properties, and how (not) to use them. I'll begin with the dumbest one: electronic codebook.

### The Electronic Codebook (ECB) Mode

The simplest of the block cipher encryption modes is electronic codebook (ECB), which is barely a mode of operation at all. ECB takes plaintext blocks $P_1, P_2, \ldots, P_N$ and processes each independently by computing $C_1 = \mathbf{E}(K, P_1)$, $C_2 = \mathbf{E}(K, P_2)$, and so on, as shown in Figure 4-6. It's a simple operation but also an insecure one. I repeat: ECB is insecure and you should not use it!

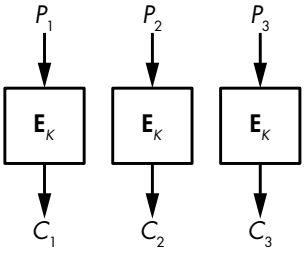


*Figure 4-6: The ECB mode*

Marsh Ray, a cryptographer at Microsoft, once said, "Everybody knows ECB mode is bad because we can see the penguin." He was referring to a famous illustration of ECB's insecurity

that uses an image of Linux's mascot, Tux, as shown in Figure 4-7. You can see the original image of Tux on the left, and the image encrypted in ECB mode using AES (though the underlying cipher doesn't matter) on the right. It's easy to see the penguin's shape in the encrypted version because all the blocks of one shade of gray in the original image are encrypted to the same new shade of gray in the new image; in other words, ECB encryption just gives you the same image but with different colors.

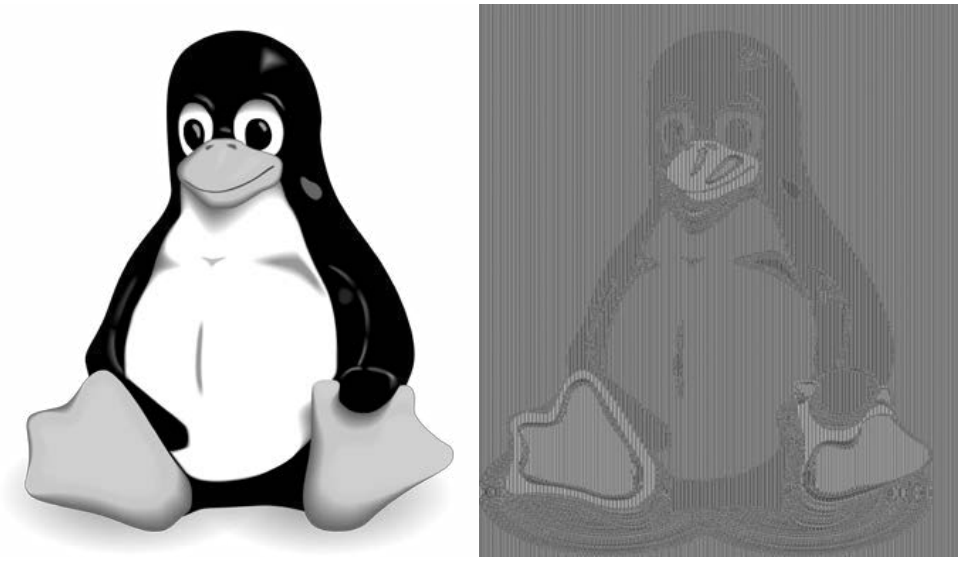


*Figure 4-7: The original image (left) and the ECB-encrypted image (right)*

The Python program in Listing 4-4 also shows ECB's insecurity. It picks a pseudorandom key and encrypts a 32-byte message `p` containing two blocks of null bytes. Notice that encryption yields two identical blocks and that repeating encryption with the same key and the same plaintext yields the same two blocks again.

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from binascii import hexlify as hexa
from os import urandom

BLOCKLEN = 16
def blocks(data):
    split = [hexa(data[i:i+BLOCKLEN]) for i in range(0, len(data), BLOCKLEN)]
    return ' '.join(split)

k = urandom(16)
print "k = %s" % hexa(k)

# create an instance of AES-128 to encrypt and decrypt
cipher = Cipher(algorithms.AES(k), modes.ECB(), backend=default_backend())
aes_encrypt = cipher.encryptor()
```

```
# set plaintext block p to the all-zero string
p = '\x00'*BLOCKLEN*2

# encrypt plaintext p to ciphertext c
c = aes_encrypt.update(p) + aes_encrypt.finalize()
print "enc(%s) = %s" % (blocks(p), blocks(c))
```

*Listing 4-4: Using AES in ECB mode in Python*

Running this script gives ciphertext blocks like this, for example:

```
$ ./aes_ecb.py
k = 50a0ebeff8001250e87d31d72a86e46d
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
5eb4b7af094ef7aca472bbd3cd72f1ed 5eb4b7af094ef7aca472bbd3cd72f1ed
```

As you can see, when the ECB mode is used, identical ciphertext blocks reveal identical plaintext blocks to an attacker, whether those are blocks within a single ciphertext or across different ciphertexts. This shows that block ciphers in ECB mode aren't semantically secure.

Another problem with ECB is that it only takes complete blocks of data, so if blocks were 16 bytes, as in AES, you could only encrypt chunks of 16 bytes, 32 bytes, 48 bytes, or any other multiple of 16 bytes. There are a few ways to deal with this, as you'll see with the next mode, CBC. (I won't tell you how these tricks work with ECB because you shouldn't be using ECB in the first place.)

## The Cipher Block Chaining (CBC) Mode

Cipher block chaining (CBC) is like ECB but with a small twist that makes a big difference: instead of encrypting the $i$th block, $P_i$, as $C_i = \mathbf{E}(K, P_i)$, CBC sets $C_i = \mathbf{E}(K, P_i \oplus C_{i-1})$, where $C_{i-1}$ is the previous ciphertext block—thereby *chaining* the blocks $C_{i-1}$ and $C_i$. When encrypting the first block, $P_1$, there is no previous ciphertext block to use, so CBC takes a random initial value (IV), as shown in Figure 4-8.

The CBC mode makes each ciphertext block dependent on all the previous blocks, and ensures that identical plaintext blocks won't be identical ciphertext blocks. The random initial value guarantees that two identical plaintexts will encrypt to distinct ciphertexts when calling the cipher twice with two distinct initial values.

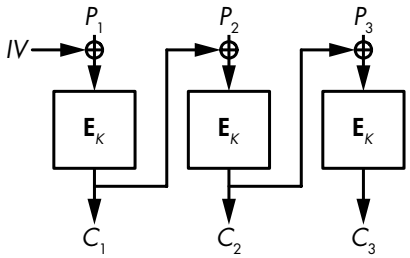



*Figure 4-8: The CBC mode*

Listing 4-5 illustrates these two benefits. This program takes an all-zero, 32-byte message (like the one in Listing 4-4), encrypts it twice with CBC, and shows the two ciphertexts. The line `iv = urandom(16)`, shown in bold, picks a new random IV for each new encryption.

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from binascii import hexlify as hexa
from os import urandom

BLOCKLEN = 16
# the blocks() function splits a data string into space-separated blocks
def blocks(data):
    split = [hexa(data[i:i+BLOCKLEN]) for i in range(0, len(data), BLOCKLEN)]
    return ' '.join(split)
k = urandom(16)
print "k = %s" % hexa(k)
# pick a random IV
iv = urandom(16)
print "iv = %s" % hexa(iv)
# pick an instance of AES in CBC mode
aes = Cipher(algorithms.AES(k), modes.CBC(iv), backend=default_backend()).encryptor()

p = '\x00'*BLOCKLEN*2
c = aes.update(p) + aes.finalize()
print "enc(%s) = %s" % (blocks(p), blocks(c))
# now with a different IV and the same key
iv = urandom(16)
print "iv = %s" % hexa(iv)
aes = Cipher(algorithms.AES(k), modes.CBC(iv), backend=default_backend()).encryptor()
c = aes.update(p) + aes.finalize()
print "enc(%s) = %s" % (blocks(p), blocks(c))
```

*Listing 4-5: Using AES in CBC mode*

The two plaintexts are the same (two all-zero blocks), but the encrypted blocks should be distinct, as in this example execution:

```
$ ./aes_cbc.py
k = 9cf0d31ad2df24f3cbbefc1e6933c872
iv = 0a75c4283b4539c094fc262aff0d17af
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
370404dcab6e9ecbc3d24ca5573d2920 3b9e5d70e597db225609541f6ae9804a
iv = a6016a6698c3996be13e8739d9e793e2
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
655e1bb3e74ee8cf9ec1540afd8b2204 b59db5ac28de43b25612dfd6f031087a
```

Alas, CBC is often used with a constant IV instead of a random one, which exposes identical plaintexts and plaintexts that start with identical blocks. For example, say the two-block plaintext $P_1 \parallel P_2$ is encrypted in CBC mode to the two-block ciphertext $C_1 \parallel C_2$. If $P_1 \parallel P_2'$ is encrypted with the same IV, where $P_2'$ is some block distinct from $P_2$, then the ciphertext will

look like $C_1 \mid\mid C_2'$, with $C_2'$ different from $C_2$ but with the same first block $C_1$. Thus, an attacker can guess that the first block is the same for both plaintexts, even though they only see the ciphertexts.

**NOTE** *In CBC mode, decryption needs to know the IV used to encrypt, so the IV is sent along with the ciphertext, in the clear.*

With CBC, decryption can be much faster than encryption due to parallelism. While encryption of a new block, $P_i$, needs to wait for the previous block, $C_{i-1}$, decryption of a block computes $P_i = \mathbf{D}(K, C_i) \oplus C_{i-1}$, where there's no need for the previous plaintext block, $P_{i-1}$. This means that all blocks can be decrypted in parallel simultaneously, as long as you also know the previous ciphertext block, which you usually do.

## How to Encrypt Any Message in CBC Mode

Let's circle back to the block termination issue and look at how to process a plaintext whose length is not a multiple of the block length. For example, how would we encrypt an 18-byte plaintext with AES-CBC when blocks are 16 bytes? What do we do with the two bytes left? We'll look at two widely used techniques to deal with this problem. The first one, padding, makes the ciphertext a bit longer than the plaintext, while the second one, *ciphertext stealing*, produces a ciphertext of the same length as the plaintext.

### Padding a Message

Padding is a technique that allows you to encrypt a message of any length, even one smaller than a single block. Padding for block ciphers is specified in the PKCS#7 standard and in RFC 5652, and is used almost everywhere CBC is used, such as in some HTTPS connections.

Padding is used to expand a message to fill a complete block by adding extra bytes to the plaintext. Here are the rules for padding 16-byte blocks:

- If there's one byte left—for example, if the plaintext is 1 byte, 17 bytes, or 33 bytes long—pad the message with 15 bytes 0f (15 in decimal).
- If there are two bytes left, pad the message with 14 bytes 0e (14 in decimal).
- If there are three bytes left, pad the message with 13 bytes 0d (13 in decimal).

If there are 15 plaintext bytes and a single byte missing to fill a block, padding adds a single 01 byte. If the plaintext is already a multiple of 16, the block length, add 16 bytes 10 (16 in decimal). You get the idea. The trick generalizes to any block length up to 255 bytes (for larger blocks, a byte is too small to encode values greater than 255).

Decryption of a padded message works like this:

1. Decrypt all the blocks as with unpadded CBC.
2. Make sure that the last bytes of the last block conform to the padding rule: that they finish with at least one 01 byte, at least two 02 bytes, or at least three 03 bytes, and so on. If the padding isn't valid—for example, if the last bytes are 01 02 03—the message is rejected. Otherwise, decryption strips the padding bytes and returns the plaintext bytes left.

One downside of padding is that it makes ciphertext longer by at least one byte and at most a block.

### Ciphertext Stealing

Ciphertext stealing is another trick used to encrypt a message whose length isn't a multiple of the block size. Ciphertext stealing is more complex and less popular than padding, but it offers at least three benefits:

- Plaintexts can be of any *bit* length, not just bytes. You can, for example, encrypt a message of 131 bits.
- Ciphertexts are exactly the same length as plaintexts.
- Ciphertext stealing is not vulnerable to padding oracle attacks, powerful attacks that sometimes work against CBC with padding (as we'll see in "Padding Oracle Attacks" on page 74).

In CBC mode, ciphertext stealing extends the last incomplete plaintext block with bits from the previous ciphertext block, and then encrypts the resulting block. The last, incomplete ciphertext block is made up of the first bits from the previous ciphertext block; that is, the bits that have not been appended to the last plaintext block.

In Figure 4-9, we have three blocks, where the last block, $P_3$, is incomplete (represented by a zero). $P_3$ is XORed with the last bits from the previous ciphertext block, and the encrypted result is returned as $C_2$. The last ciphertext block, $C_3$, then consists of the first bits from the previous ciphertext block. Decryption is simply the inverse of this operation.

There aren't any major problems with ciphertext stealing, but it's inelegant and hard to get right, especially when NIST's standard specifies three different ways to implement it (see Special Publication 800-38A).

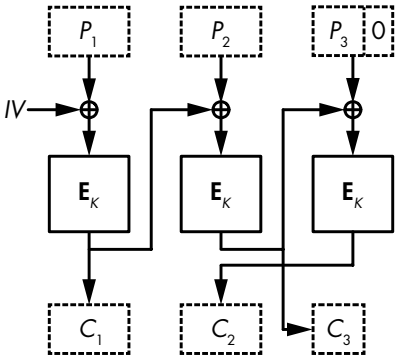



*Figure 4-9: Ciphertext stealing for CBC-mode encryption*

### The Counter (CTR) Mode

To avoid the troubles and retain the benefits of ciphertext stealing, you should use counter mode (CTR). CTR is hardly a block cipher mode: it turns a block cipher into a stream cipher that just takes bits in and spits bits out and doesn't embarrass itself with the notion of blocks. (I'll discuss stream ciphers in detail in Chapter 5.)

In CTR mode (see Figure 4-10), the block cipher algorithm won't transform plaintext data. Instead, it will encrypt blocks composed of a *counter* and a *nonce*. A counter is an integer that is incremented for each block. No two blocks should use the same counter within a message, but different messages can use the same sequence of counter values (1, 2, 3, . . .). A nonce is a number used only once. It is the same for all blocks in a single message, but no two messages should use the same nonce.

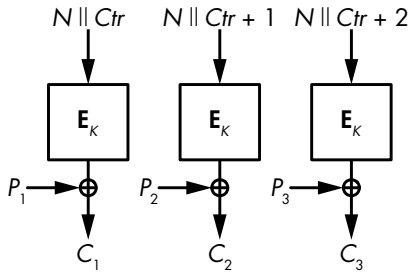



*Figure 4-10: The CTR mode*

As shown in Figure 4-10, in CTR mode, encryption XORs the plaintext and the stream taken from "encrypting" the nonce, *N*, and counter, *Ctr*. Decryption is the same, so you only need the encryption algorithm for both encryption and decryption. The Python script in Listing 4-6 gives you a hands-on example.

```
#!/usr/bin/env python

from Crypto.Cipher import AES
from Crypto.Util import Counter
from binascii import hexlify as hexa
from os import urandom
from struct import unpack

k = urandom(16)
print "k = %s" % hexa(k)

# pick a starting value for the counter
nonce = unpack('<Q', urandom(8))[0]
# instantiate a counter function
ctr = Counter.new(128, initial_value=nonce)

# pick an instance of AES in CTR mode, using ctr as counter
aes = AES.new(k, AES.MODE_CTR, counter=ctr)

# no need for an entire block with CTR
p = '\x00\x01\x02\x03'

# encrypt p
c = aes.encrypt(p)
print "enc(%s) = %s" % (hexa(p), hexa(c))
```

```
# decrypt using the encrypt function
ctr = Counter.new(128, initial_value=nonce)
aes = AES.new(k, AES.MODE_CTR, counter=ctr)
p = aes.encrypt(c)
print "enc(%s) = %s" % (hexa(c), hexa(p))
```

*Listing 4-6: Using AES in CTR mode*

The example execution encrypts a 4-byte plaintext and gets a 4-byte ciphertext. It then decrypts that ciphertext using the encryption function:

```
$ ./aes_ctr.py
k = 130a1aa77fa58335272156421cb2a3ea
enc(00010203) = b23d284e
enc(b23d284e) = 00010203
```

As with the initial value in CBC, CTR's nonce is supplied by the encrypter and sent with the ciphertext in the clear. But unlike CBC's initial value, CTR's nonce doesn't need to be random, it simply needs to be unique. A nonce should be unique for the same reason that a one-time pad shouldn't be reused: when calling the pseudorandom stream, $S$, if you encrypt $P_1$ to $C_1 = P_1 \oplus S$ and $P_2$ to $C_2 = P_2 \oplus S$ using the same nonce, then $C_1 \oplus C_2$ reveals $P_1 \oplus P_2$.

A random nonce will do the trick only if it's long enough; for example, if the nonce is $n$ bits, chances are that after $2^{n/2}$ encryptions and as many nonces you'll run into duplicates. Sixty-four bits are therefore insufficient for a random nonce, since you can expect a repetition after approximately $2^{32}$ nonces, which is an unacceptably low number.

The counter is guaranteed unique if it's incremented for every new plaintext, and if it's long enough; for example, a 64-bit counter.

One particular benefit to CTR is that it can be faster than in any other mode. Not only is it parallelizable, but you can also start encrypting even before knowing the message by picking a nonce and computing the stream that you'll later XOR with the plaintext.

## How Things Can Go Wrong

There are two must-know attacks on block ciphers: meet-in-the-middle attacks, a technique discovered in the 1970s but still used in many cryptanalytic attacks (not to be confused with man-in-the-middle attacks), and padding oracle attacks, a class of attacks discovered in 2002 by academic cryptographers, then mostly ignored, and finally rediscovered a decade later along with several vulnerable applications.

### Meet-in-the-Middle Attacks

The 3DES block cipher is an upgraded version of the 1970s standard DES that takes a key of $56 \times 3 = 168$ bits (an improvement on DES's 56-bit key). But the security level of 3DES is 112 bits instead of 168 bits, because of the *meet-in-the-middle (MitM)* attack.

As you can see in Figure 4-11, 3DES encrypts a block using the DES encryption and decryption functions: first encryption with a key, $K_1$, then decryption with a key, $K_2$, and finally encryption with another key, $K_3$. If $K_1 = K_2$, the first two calls cancel themselves out and 3DES boils down to a single DES with key $K_3$. 3DES does encrypt-decrypt-encrypt rather than encrypting thrice to allow systems to emulate DES when necessary using the new 3DES interface.

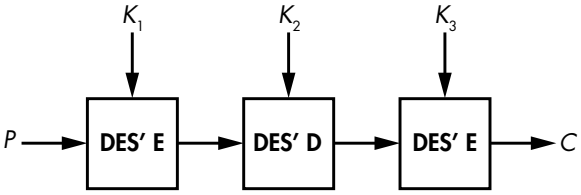



*Figure 4-11: The 3DES block cipher construction*

Why use triple DES and not just double DES, that is, $\mathbf{E}(K_2, \mathbf{E}(K_1, P))$? It turns out that the MitM attack makes double DES only as secure as single DES. Figure 4-12 shows the MitM attack in action.

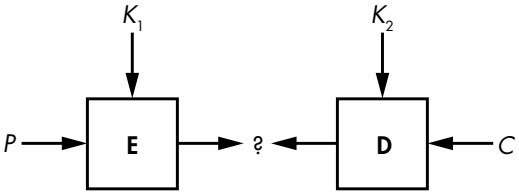



*Figure 4-12: The meet-in-the-middle attack*

The meet-in-the-middle attack works as follows to attack double DES:

1. Say you have $P$ and $C = \mathbf{E}(K_2, \mathbf{E}(K_1, P))$ with two unknown 56-bit keys, $K_1$ and $K_2$. (DES takes 56-bit keys, so double DES takes 112 key bits in total.) You build a key–value table with $2^{56}$ entries of $\mathbf{E}(K_1, P)$, where $\mathbf{E}$ is the DES encryption function and $K_1$ is the value stored.
2. For all $2^{56}$ values of $K_2$, compute $\mathbf{D}(K_2, C)$ and check whether the resulting value appears in the table as an index (thus as a middle value, represented by a question mark in Figure 4-12).
3. If a middle value is found as an index of the table, you fetch the corresponding $K_1$ from the table and verify that the $(K_1, K_2)$ found is the right one by using other pairs of $P$ and $C$. Encrypt $P$ using $K_1$ and $K_2$ and then check that the ciphertext obtained is the given $C$.

This method recovers $K_1$ and $K_2$ by performing about $2^{57}$ instead of $2^{112}$ operations: step 1 encrypts $2^{56}$ blocks and then step 2 decrypts at most $2^{56}$ blocks, for $2^{56} + 2^{56} = 2^{57}$ operations in total. You also need to store $2^{56}$ elements of 15 bytes each, or about 1 exabyte. That's a lot, but there's a trick that allows you to run the same attack with only negligible memory (as you'll see in Chapter 6).

As you can see, you can apply the MitM attack to 3DES in almost the same way you would to double DES, except that the third stage will go through all $2^{112}$ values of $K_2$ and $K_3$. The whole attack thus succeeds after performing about $2^{112}$ operations, meaning that 3DES gets only 112-bit security despite having 168 bits of key material.

## Padding Oracle Attacks

Let's conclude this chapter with one of the simplest and yet most devastating attacks of the 2000s: the padding oracle attack. Remember that padding fills the plaintext with extra bytes in order to fill a block. A plaintext of 111 bytes, for example, is a sequence of six 16-byte blocks followed by 15 bytes. To form a complete block, padding adds a 01 byte. For a 110-byte plaintext, padding adds two 02 bytes, and so on.

A *padding oracle* is a system that behaves differently depending on whether the padding in a CBC-encrypted ciphertext is valid. You can see it as a black box or an API that returns either a *success* or an *error* value. A padding oracle can be found in a service on a remote host sending error messages when it receives malformed ciphertexts. Given a padding oracle, padding oracle attacks record which inputs have a valid padding and which don't, and exploit this information to decrypt chosen ciphertext values.

Say you want to decrypt ciphertext block $C_2$. I'll call $X$ the value you're looking for, namely $\mathbf{D}(K, C_2)$, and $P_2$ the block obtained after decrypting in CBC mode (see Figure 4-13). If you pick a random block $C_1$ and send the two-block ciphertext $C_1 \mid\mid C_2$ to the oracle, decryption will only succeed if $C_1 \oplus P_2 = X$ ends with valid padding—a single 01 byte, two 02 bytes, or three 03 bytes, and so on.

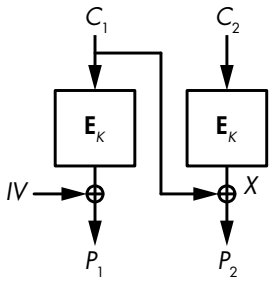



*Figure 4-13: Padding oracle attacks recover X by choosing $C_1$ and checking the validity of padding.*

Based on this observation, padding oracle attacks on CBC encryption can decrypt a block $C_2$ like this (bytes are denoted in array notation: $C_1[0]$ is $C_1$'s first byte, $C_1[1]$ its second byte, and so on up to $C_1[15]$, $C_1$'s last byte):

1. Pick a random block $C_1$ and vary its last byte until the padding oracle accepts the ciphertext as valid. Usually, in a valid ciphertext, $C_1[15] \oplus X[15] = 01$, so you'll find $X[15]$ after trying around 128 values of $C_1[15]$.
2. Find the value $X[14]$ by setting $C_1[15]$ to $X[15] \oplus 02$ and searching for the $C_1[14]$ that gives correct padding. When the oracle accepts the ciphertext as valid, it means you have found $C_1[14]$ such that $C_1[14] \oplus X[14] = 02$.
3. Repeat steps 1 and 2 for all 16 bytes.

The attack needs on average 128 queries to the oracle for each of the 16 bytes, which is about 2000 queries in total. (Note that each query must use the same initial value.)

**NOTE** *In practice, implementing a padding oracle attack is a bit more complicated than what I've described, because you have to deal with wrong guesses at step 1. A ciphertext may have valid padding not because* $P_2$ *ends with a single 01 but because it ends with two 02 bytes or three 03 bytes. But that's easily managed by testing the validity of ciphertexts where more bytes are modified.*

## Further Reading

There's a lot to say about block ciphers, be it in how algorithms work or in how they can be attacked. For instance, Feistel networks and SPNs aren't the only ways to build a block cipher. The block ciphers IDEA and FOX use the Lai–Massey construction, and Threefish uses ARX networks, a combination of addition, word rotations, and XORs.

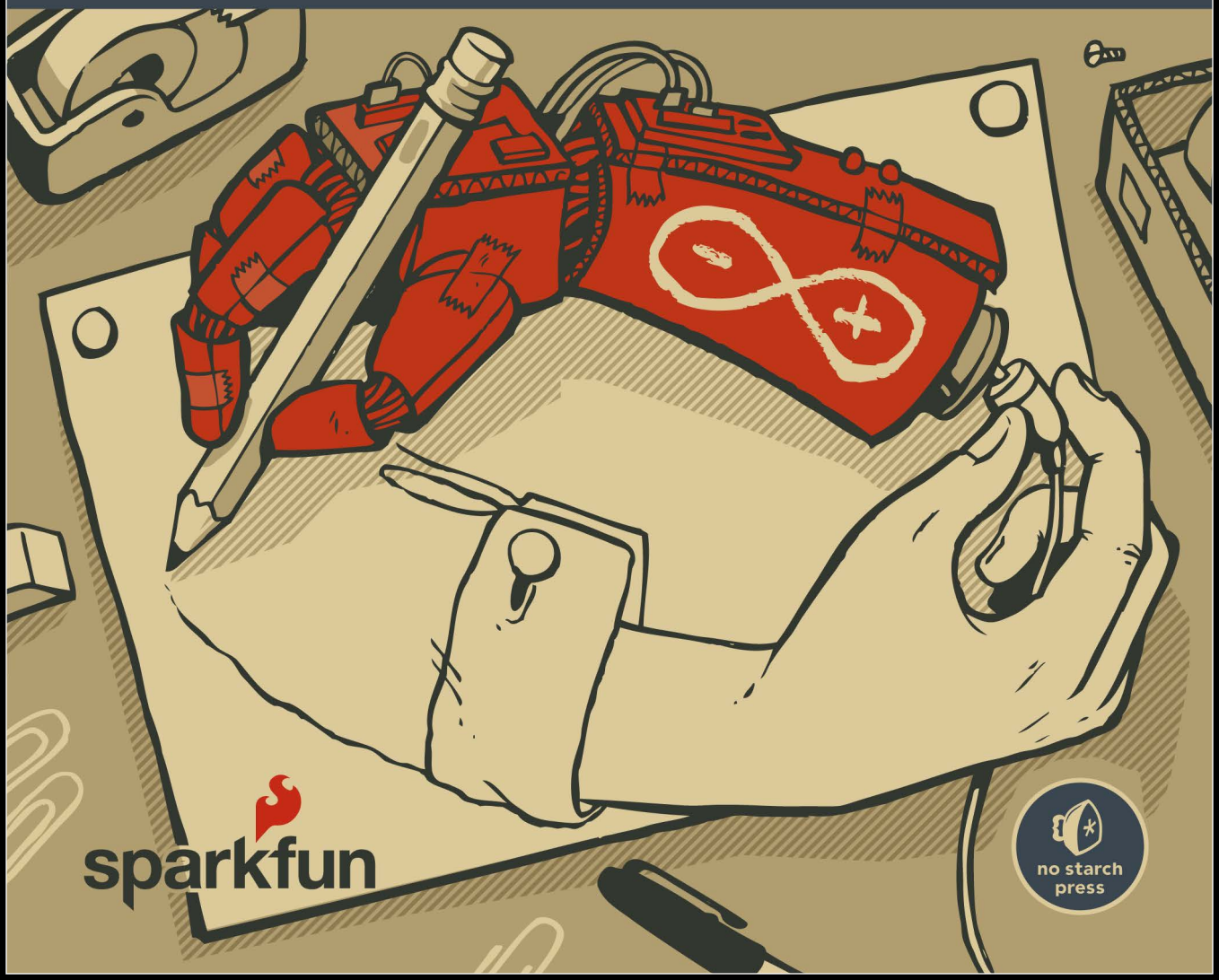
There are also many more modes than just ECB, CBC, and CTR. Some modes are folklore techniques that nobody uses, like CFB and OFB, while others are for specific applications, like XTS for tweakable encryption or GCM for authenticated encryption.

I've discussed Rijndael, the AES winner, but there were 14 other algorithms in the race: CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, Magenta, MARS, RC6, SAFER+, Serpent, and Twofish. I recommend that you look them up to see how they work, how they were designed, how they have been attacked, and how fast they are. It's also worth checking out the NSA's designs (Skipjack, and more recently, SIMON and SPECK) and more recent "lightweight" block ciphers such as KATAN, PRESENT, or PRINCE.

# THE ARDUINO INVENTOR'S GUIDE

## LEARN ELECTRONICS BY MAKING 10 AWESOME PROJECTS

BRIAN HUANG AND DEREK RUNBERG



sparkfun

no starch press


2

# A STOPLIGHT FOR YOUR HOUSE

IN YOUR FIRST BIG STEP TOWARD WORLD DOMINATION THROUGH EMBEDDED ELECTRONICS, YOU SET UP THE ARDUINO IDE AND BLINKED AN LED. THAT'S HUGE, BUT WITH AN ARDUINO, NO PROJECT NEEDS TO STOP AT JUST ONE LED. THIS PROJECT WILL SHOW YOU HOW TO EXPAND YOUR FIRST LED SKETCH TO DISPLAY

a blinking pattern on *three* LEDs. Your mission, should you choose to accept it, is to build and program a stoplight for a busy hallway in your house (see Figure 2-1).

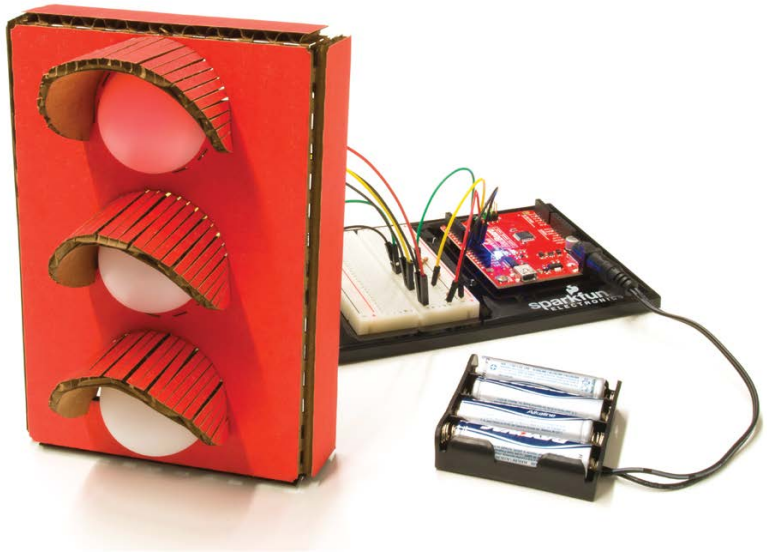


**FIGURE 2-1:**
The completed Stoplight project

## MATERIALS TO GATHER

The materials in this project are all pretty simple. All of the electronic parts are standard in the SparkFun Inventor's Kit, except for the ones marked with an asterisk (*). If you're using your own kit or piecing together parts yourself, see the following parts list. Figure 2-2 shows all of the parts used in this project.

### Electronic Parts

- One SparkFun RedBoard (DEV-13975), Arduino Uno (DEV-11021), or any other Arduino-compatible board
- One USB Mini-B cable (CAB-11301 or your board's USB cable; not shown)
- One solderless breadboard (PRT-12002)
- One red LED, one yellow LED, and one green LED (COM-12062)
- Three 330 Ω resistors (COM-08377, or COM-11507 for a pack of 20)
- Male-to-male jumper wires (PRT-11026)
- Male-to-female jumper wires (PRT-09140*)
- (Optional) One 4 AA battery holder (PRT-09835*; not shown)

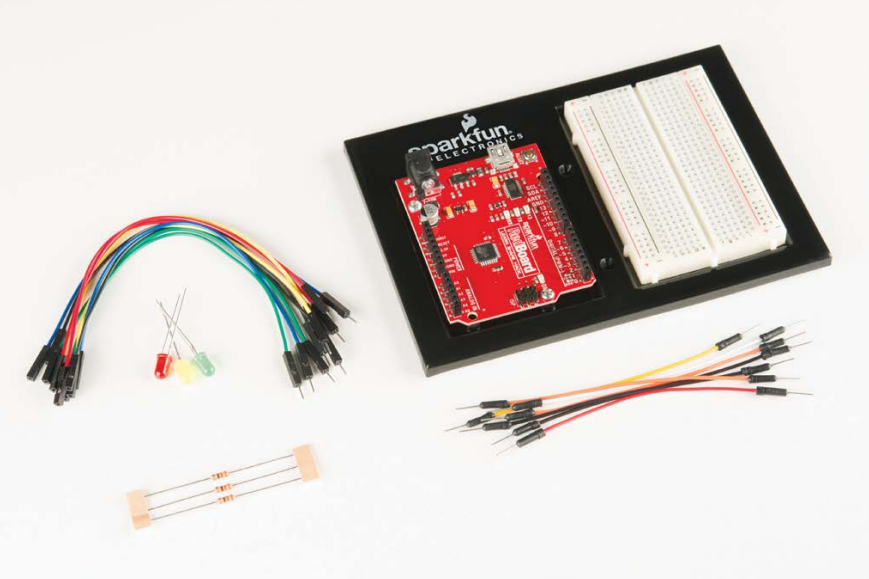



**FIGURE 2-2:**
Components for the Stoplight

## Other Materials and Tools

If you want to build an enclosure like the one in Figure 2-1 or follow the suggestions in "Going Further" on page 64, you'll also need the following supplies, shown in Figures 2-3 and 2-4:

- Pencil
- Craft knife
- Metal ruler
- Pliers
- Wire stripper
- Glue (hot glue gun or craft glue)
- (Optional) Drill and a 3/16-inch drill bit
- (Optional) Soldering iron
- (Optional) Solder
- (Optional) Helping hands (not shown)
- Cardboard (about 12 inches square) or a cardboard box
- Two ping-pong balls
- Enclosure template (see Figure 2-15 on page 55)

**NOTE**

*Good, clean cardboard will be worth its weight in gold in these projects. We suggest picking up cardboard sheets from a craft or art supply store.*

**FIGURE 2-3:**
Recommended tools


**FIGURE 2-4:**
Recommended building materials


## NEW COMPONENT: THE RESISTOR

Although you used an LED on its own in Project 1, in most cases it's best to use a *resistor* to protect the LED from too much current. Resistors like the ones in Figure 2-5 are everywhere. They are indispensable when you're building circuits, and you'll need them to complete this project, too.

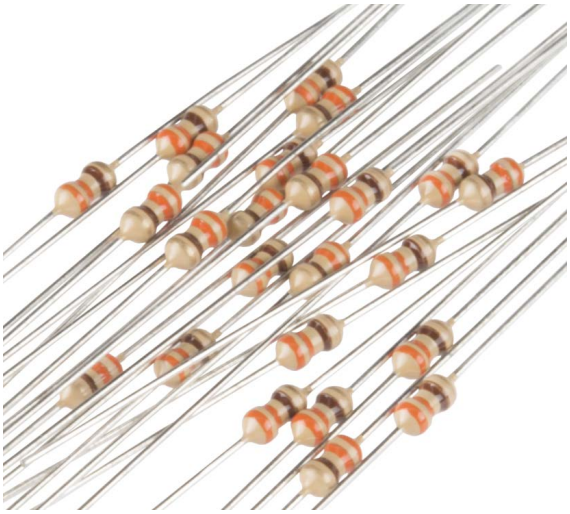


**FIGURE 2-5:**
Resistors up close and personal

If you think of electricity like the flow of water through a pipe, a resistor is analogous to a point where the pipe size narrows, reducing the water flow. (If you're curious, see "Visualizing Electricity as Water in a Pipe" on page 4, which describes this metaphor in detail.) Resistors control or limit the flow of current.

Resistance is measured in *ohms* (typically shortened to Ω, the Greek symbol omega), and the colored bands on resistors represent their resistance. You'll find a resistor color band decoder in "Resistors and Bands" on page 308; however, in this book, you only need to be able to identify two different values of resistors: 330 Ω and 10 kΩ. The bands on a 330 Ω resistor are orange, orange, and brown (see Figure 2-5), while on a 10 kΩ resistor they're brown, black, and orange. There is also a fourth band on a resistor, and its color indicates the resistor's *tolerance*. A resistor's value will be accurate within a certain tolerance: silver means the resistor has a 5 percent tolerance, while gold indicates a 10 percent tolerance. The projects in this book aren't sensitive enough for the tolerance level to make a difference, though, so we'll just refer to the resistors by their assumed value, which will work for either tolerance band.

Some components, like LEDs, can be damaged if the current flowing to them is too high, and resistors can protect those components by reducing the current. Having a resistor in line with an LED to limit the current to a safe level is a good precaution so your LED doesn't burn out—or, in the worst case, pop! (Yes, they can literally pop.) From here on, we'll use current-limiting resistors in all projects.

### WHY THE STOPLIGHT USES 330 Ω RESISTORS

An average red LED has a maximum current rating of about 20 mA, as listed on its datasheet. In order to protect it, you need to add a resistor to keep the current below this limit. But how do you know to use a 330 Ω resistor?

The output pins on the Arduino provide 5 V when they are turned on. Depending on the color, each LED needs a slightly different amount of voltage to turn on, typically in the range of 2.0 to 3.5 V. A red LED turns on at about 2 V, and that leaves 3 V remaining. The 3 V will be dissipated across a resistor or anything else that is in line in the circuit. It's generally good practice to limit the current going through an LED to about half the maximum, so for the red LED with a maximum current rating of 20 mA, you get 10 mA. You can calculate the resistor needed for 3 V and 10 mA with *Ohm's law* (remember 10 mA = 0.01 A):

$$V = I \times R$$

$$R = \frac{V}{I} = \frac{3\text{ V}}{0.01\text{ A}} = 300\ \Omega$$

But 300 Ω isn't a standard resistor value. The closest standard resistor value is 330 Ω, and usually the nearest standard resistor is good enough. This should ensure that the LED lasts for a very, very long time. Since the resistor will be dictating the current, this is a *current-limiting resistor*.

If you have different resistors available, you could use a different value resistor and see what happens. Bigger resistors will make the current smaller, and smaller resistors will make the current bigger. What happens if you use the 10 kΩ resistor instead?

## BUILD THE STOPLIGHT PROTOTYPE

Now it's time to build the circuit. First, take a look at the schematic shown in Figure 2-6. You'll build this on a breadboard, as shown in Figure 2-7.

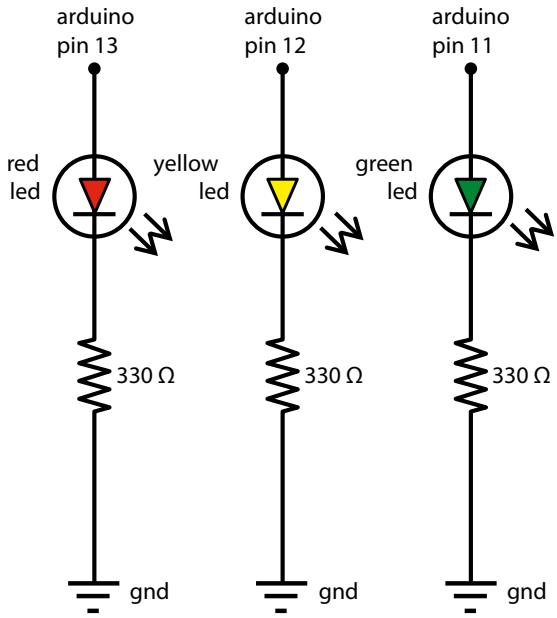



**FIGURE 2-6:**
Schematic diagram for the Stoplight project

The schematic illustrates how each component is connected electrically. Pin 13, pin 12, and pin 11 on the Arduino will each be used to control an individual LED on the Stoplight circuit. As you can see in the schematic, each LED is connected to an individual resistor, and each resistor is connected to GND (ground). Next, let's look at the wiring.

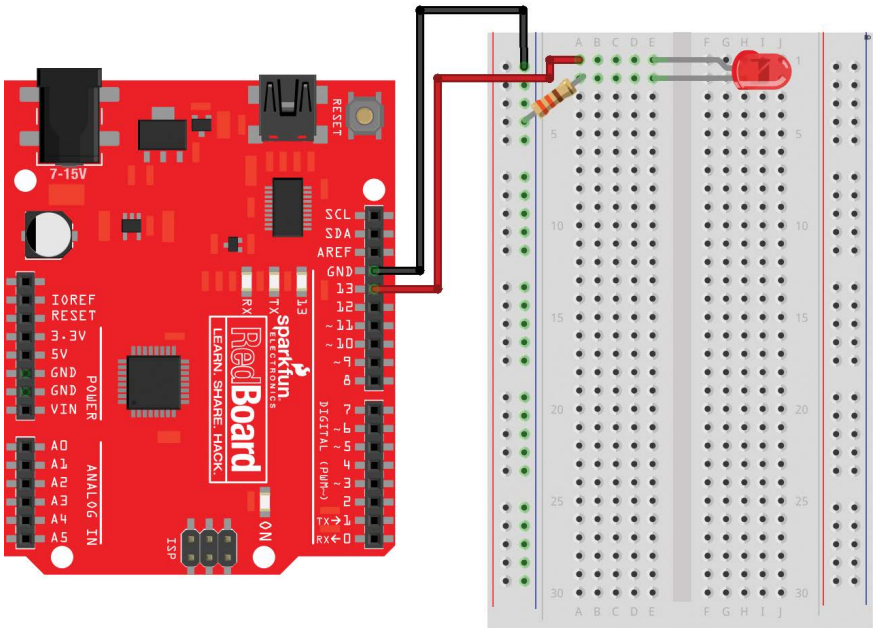



**FIGURE 2-7:**
Connecting a red LED to a breadboard with a current-limiting resistor

## Connect the Red LED to the Breadboard

Now you'll start to translate the schematic into an actual circuit. In the first project, you blinked an LED built into the Arduino board. This LED was internally wired to pin 13 on the Arduino. Because you'll be using three discrete LEDs, you need to wire these up yourself. Take out your breadboard, and, following the schematic in Figure 2-6 or the illustrated diagram in Figure 2-7, connect pin 13 to the positive (long) leg of the LED.

**NOTE**

*For a refresher on how breadboards work, see "Prototyping Circuits" on page 6.*

To wire this on the breadboard, we suggest that you first position your Arduino and breadboard as shown in Figure 2-7. (This will be the standard layout throughout the book.) Then, find a red LED and a 330 Ω resistor. Bend the resistor legs as shown in Figure 2-8 so that the resistor is easier to insert into the breadboard. We suggest using wire cutters to trim both resistor legs by about half their length to make the resistor easier to work with. Resistors aren't polarized like LEDs, so you don't have to keep track of which leg is positive or negative.

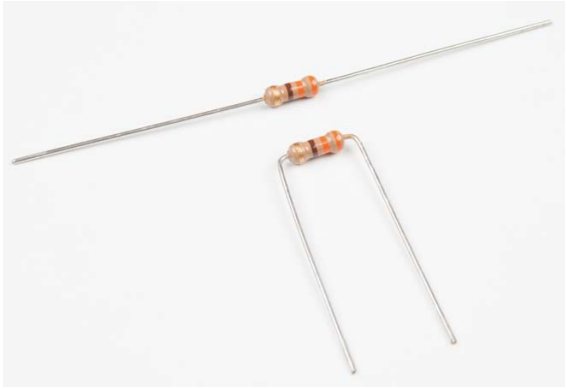


**FIGURE 2-8:** Bending a resistor

Figure 2-9 shows a diagram of a typical breadboard. Most breadboards have labeled columns and numbered rows as references. Using these reference points, insert the LED into your breadboard as shown in Figure 2-7. The long, positive leg (anode) should be in column E, row 1 (E1) on the breadboard, and the short, negative leg (cathode) should be in column E, row 2 (E2). Now, find a 330 Ω (orange-orange-brown) resistor. Insert one leg of the resistor into any hole in row 2 of the breadboard to connect the resistor to the short leg of the LED. In our diagram, we insert this leg of the resistor into A2 on the breadboard. On all standard breadboards, for each row, columns A–E are connected, and columns F–J are connected. Now, insert the other leg of the resistor into the breadboard's *negative power rail*, which is the column marked with a blue or black line and a – (minus) symbol.

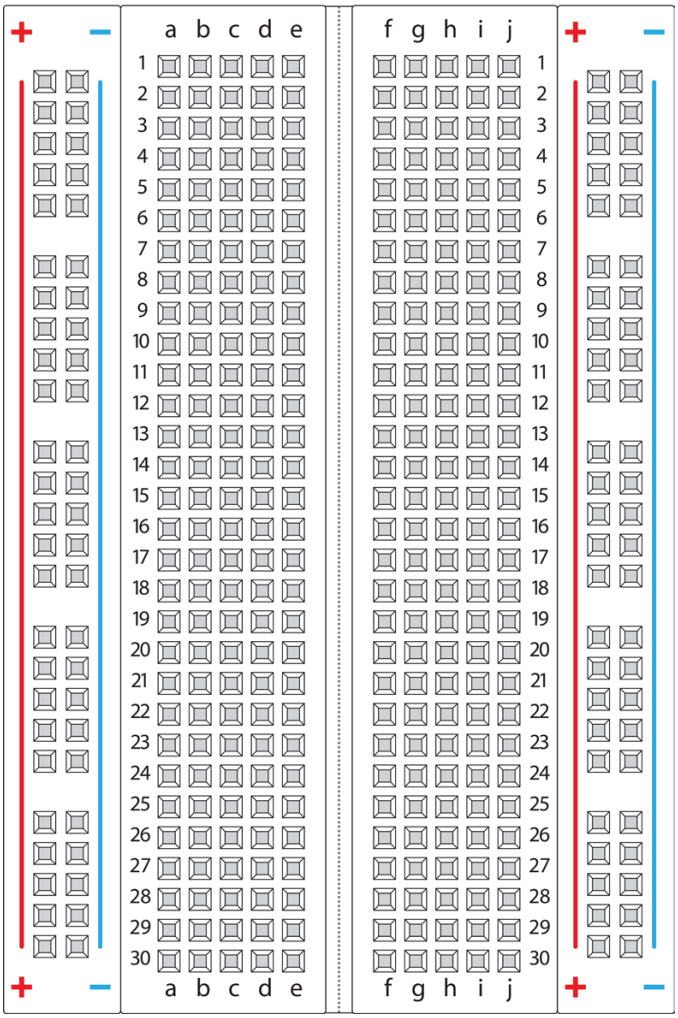



**FIGURE 2-9:**
A breadboard has numbered rows and columns labeled with letters.

## Add Power to the Breadboard

Grab two male-to-male jumper wires. We suggest using black for ground (GND) and red for power, and that's the convention we'll follow throughout this book.

Connect the black wire from the GND pin on the Arduino to the negative power rail on the breadboard. There are three pins labeled GND on the Arduino. You can use any of these. The power for each LED will actually come from the digital pins. Since pin 13 will power the red LED, connect a wire from pin 13 on the Arduino to A1 on the breadboard.

Plug your Arduino board into your computer using a USB cable, and the "Hello, world!" sketch from Project 1 should run, causing your LED to blink. In fact, both the LED on the breadboard and the LED on the Arduino should be blinking, because they're both wired into pin 13.

If the breadboard LED doesn't blink but the Arduino one does, double-check your wiring and the orientation of the LED. Make sure that the shorter leg is in the second row of the breadboard, connected to the resistor, and that the resistor is connected to GND through the negative power rail. After you get the red LED blinking, disconnect the Arduino from the computer so that you can safely build the rest of the circuit. It's best practice to disconnect the board while building your circuit.

## Add the Yellow and Green LEDs

Now, connect the yellow LED to pin 12 on the Arduino and the green LED to pin 11; you can follow the same basic instructions you followed for the red LED, but use different pairs of rows for each new LED, as in the final wiring diagram in Figure 2-10.

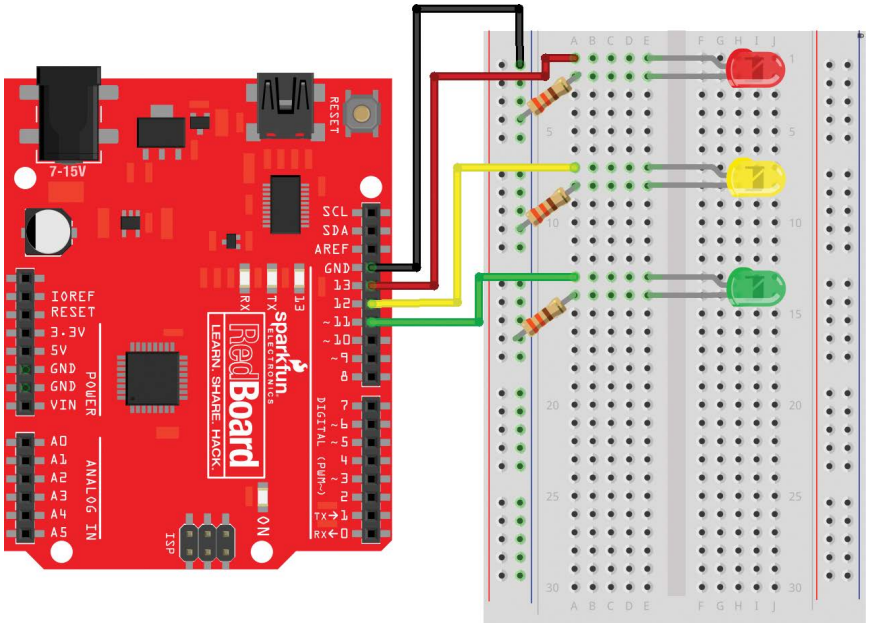



**FIGURE 2-10:** The final Stoplight circuit, using pins 11, 12, and 13

Each LED should have its own resistor wired to the ground rail, just like the schematic from Figure 2-6. Notice, too, that we gave each LED a little space on the breadboard so that we could have room to plug in wires without messing up other parts of the circuit. Although we suggested a specific way to plug in this circuit, remember that you can use any part of the breadboard—so long as the two wires you're trying to connect are in the same row. Once you're done, your circuit should resemble Figure 2-11.

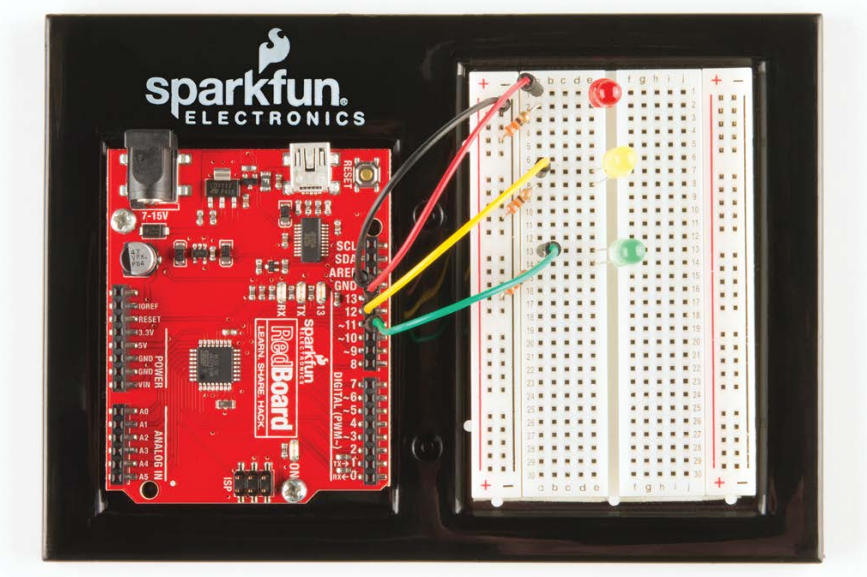



**FIGURE 2-11:**
The completed Stoplight circuit, including the Arduino, LEDs, and resistors

To mimic a real stoplight, this project needs a way to turn on each light for a certain amount of time and then switch to the next one. Fortunately, an Arduino sketch can use all kinds of instructions, including timing commands, to control a circuit.

## PROGRAM THE STOPLIGHT

Now, plug your Arduino back into your computer. It's time to get programming! Open the Arduino IDE to start a new sketch.

### Confirm Your IDE Settings

When writing any sketch, you should always start with a little housekeeping. First, check that the Board type and Port are properly set. Click **Tools ▸ Board** now. If you're using the SparkFun RedBoard or a standard Arduino Uno, select **Arduino/Genuino Uno**. Then, click **Tools ▸ Port**. In Windows, your Arduino should be set to the highest numbered COM port. On OS X or Linux, the port should be listed as */dev/cu.usbserial-A<xxxx>*, where *<xxxx>* is a string of random characters unique to your Arduino.

### Create Placeholders for Pin Numbers

With your IDE settings confirmed, you're ready to create the sketch. As discussed in "Anatomy of an Arduino Sketch" on page 27, a basic Arduino sketch consists of two parts: the `setup()` function and the `loop()` function. That simplified description is true for most simple sketches, but more complex sketches have many different parts. One new part that the Stoplight sketch uses is the *global*

*namespace*, which is the part of your sketch above the `setup()` function and completely outside of any function. In this space, you can define certain names (*variables*) as placeholders for values, and these values will then be available for all parts of your sketch to use. Arduino sketches can work with several types of values.

### Data That Sketches Understand

The Arduino language includes a number of possible *data types* for values, and there are a few you'll run into often when writing sketches. The following list isn't exhaustive, but it touches on the big ones and shows how their names appear in code:

**Integer (`int`)** A whole number that ranges from –32,768 to 32,767

**Float (`float`)** A number that has a decimal point and ranges from –3.4028235E+38 to 3.4028235E+38

**Byte (`byte`)** A number that ranges from 0 to 255

**Character (`char`)** A single letter, denoted by a set of single quotes, such as `'a'`

**String (`String`)** A series of characters, denoted by a set of double quotes, such as `"hello"`

**Boolean (`Boolean`)** A value of either `true` or `false`, which maps to `1` or `0` in the sketch and `HIGH` or `LOW` in terms of pin output

Arduino sketches require you to specify the data type of a variable when you define it. Let's look at how that works.

### Values That Can Change

Most values you'll create to use in your sketches will be variables. Think of a variable as a placeholder for a piece of data. That data can be a number, a letter, or even a whole sentence.

Before you can use a variable, you have to *define* it, which includes giving it a name, declaring its data type, and initializing it with a value. It's a good habit to give a variable a value at the moment you define it, which looks something like this:

```
❶int ❷val = ❸10;
```

This variable definition has three parts: the data type ❶, the name of the variable ❷, and the variable's value ❸. At the end of this line, notice that there is a semicolon—this denotes the end of a statement or instruction. The semicolon is very important, and forgetting it is often the root cause of many compiler errors or bugs in code, so be careful to remember it!

When choosing a variable name, you can use any unbroken set of characters, including letters and numbers. There is one caveat here: variables cannot start with a number or consist of any special characters. We suggest making variable names as descriptive as possible, while keeping them short. It's a chance for you to be a little creative with abbreviating words and descriptions. In this example, we chose to name the variable `val` (short for *value*), and `10` is the variable's *initialized value*, or the value assigned to a variable to start with. You don't need to initialize a variable when you define it, but doing both at the same time is helpful and a good practice.

For this project, you'll create three variables to store pin numbers for the three LEDs the Arduino will control. It's a lot easier to work with a variable that describes an LED color than it is to try to remember which LED is connected to which pin!

Start a new sketch, and add the code in Listing 2-1 to the global namespace of your sketch.

```
byte redPin = 13;
byte ylwPin = 12;
byte grnPin = 11;
```

**LISTING 2-1:**
Variables that represent pin numbers

**NOTE**

*For legibility, we* camel-cased *the variable names by capitalizing the* `p` *in* `pin`*. Camel-casing is a coding convention that allows you to separate words in a variable without using spaces.*

Again, these three variables store the pin numbers for the three LEDs. On the Arduino, pin numbers are limited to whole numbers between 0 and 13, so we use the `byte` data type. We can use `byte` because we know that the pin number will be less than 255. Notice that each variable's name describes what it contains: `redPin` is for the red LED pin, `ylwPin` is the yellow LED pin, and `grnPin` is the green LED pin. And, just as Figure 2-10 shows, the red pin is pin 13, yellow is pin 12, and green is pin 11. Now, anytime you use a pin number in your sketch, you can use the descriptive variable name instead.

## Write the setup() Function

To continue writing the Stoplight sketch, add the `setup()` function in Listing 2-2.

```
void setup()
{
  //red LED
  pinMode(redPin❶, OUTPUT❷);
  //yellow LED
  pinMode(ylwPin, OUTPUT);
```

**LISTING 2-2:**
`setup()` code for the Stoplight

```
  //green LED
  pinMode(grnPin, OUTPUT);
}
```

Just like the "Hello, world!" sketch in Project 1 (see "The `setup()` Function" on page 30), this sketch configures the digital pins of the Arduino in `setup()` with the `pinMode()` function.

This project uses three different digital pins, so the sketch has three separate `pinMode()` functions. Each function call includes a pin number as its variable ❶ (`redPin`, `ylwPin`, and `grnPin`) and the constant `OUTPUT` ❷. It uses `OUTPUT` because this sketch controls LEDs, which are output devices. We'll introduce `INPUT` devices in Project 4.

## Write the loop() Function

Next comes the `loop()` function. Normal stoplights cycle from red to green to yellow and then back to red, so this project does, too. Copy the code from Listing 2-3 into the `loop()` portion of your sketch.

**LISTING 2-3:**
`loop()` code for the Stoplight

```
void loop()
{
  //red on
  digitalWrite(redPin, HIGH);
  digitalWrite(ylwPin, LOW);
  digitalWrite(grnPin, LOW);
  delay(2000);

  //green on
  digitalWrite(redPin, LOW);
  digitalWrite(ylwPin, LOW);
  digitalWrite(grnPin, HIGH);
  delay(1500);

  //yellow on
  digitalWrite(redPin, LOW);
  digitalWrite(ylwPin, HIGH);
  digitalWrite(grnPin, LOW);
  delay(500);
}
```

The Stoplight will have only one light on at a time, to avoid confusing your hallway traffic and causing chaos. To maintain order, each time an LED is turned on, the other LEDs should be turned off. For example, if you wanted the red light to be on, you'd call the function `digitalWrite(redPin, HIGH)`, followed by `digitalWrite(ylwPin, LOW)` and `digitalWrite(grnPin, LOW)`. The first call writes `HIGH` to

turn on the red LED on `redPin` (pin 13), and the other two calls write `LOW` to `ylwPin` and `grnPin` (pins 12 and 11) to turn off the yellow and green LEDs. Because the Arduino runs at 16 MHz (roughly one instruction per 16 millionth of a second), the time between these commands is on the order of a few microseconds. These three commands run so fast that you can assume they all happen at the same time. Finally, notice the function `delay(2000)`. This function pauses the sketch and keeps the red light on for 2,000 ms, or 2 seconds, before executing the next set of instructions.

The code for the yellow and green LEDs repeats the same concept, setting the corresponding pin to `HIGH` and the others to `LOW` and delaying for different lengths of time. For your own Stoplight, try changing the delay times to something a little more realistic for your hallway's traffic. Remember that the value you pass to the `delay()` function is the amount of time you want the LED to stay on in milliseconds.

## Upload the Sketch

After you've typed in all of the code, double-check that it looks like the code in Listing 2-4, save your sketch, and upload it to your Arduino by clicking **Sketch ▸ Upload** or pressing CTRL-U. If the IDE gives you any errors, double-check your code to make sure that it matches the example code exactly. Your instructions should have the same spelling, capitalization, and punctuation, and don't forget the semicolon at the end of each instruction.

When everything works, your LEDs should turn on and off in a cycle that is similar to a real stoplight—starting with a red light, followed by a green light, and then a short yellow light before returning to the top of the `loop()` function and going back to red. Your sketch should continue to run this way indefinitely while the Arduino is powered.

```
byte redPin = 13;
byte ylwPin = 12;
byte grnPin = 11;

void setup()
{
  pinMode(redPin, OUTPUT);
  pinMode(ylwPin, OUTPUT);
  pinMode(grnPin, OUTPUT);
}
```

**LISTING 2-4:**
Complete code for the Stoplight

```
void loop()
{
  //red on
  digitalWrite(redPin, HIGH);
  digitalWrite(ylwPin, LOW);
  digitalWrite(grnPin, LOW);
  delay(2000);

  //green on
  digitalWrite(redPin, LOW);
  digitalWrite(ylwPin, LOW);
  digitalWrite(grnPin, HIGH);
  delay(1500);

  //yellow on
  digitalWrite(redPin, LOW);
  digitalWrite(ylwPin, HIGH);
  digitalWrite(grnPin, LOW);
  delay(500);
}
```

## Make the Stoplight Portable

When your Arduino is connected to your computer, it's receiving power through the USB port. But what if you want to move your project or show it around? You'll need to add a portable power source—namely, a battery pack. The Arduino board has a barrel jack power port for plugging battery packs into, as well as an on-board voltage regulator that will accept any voltages from about 6 V to 18 V. There are many different battery adapters available, but we like using a 4 AA battery adapter for a lot of our projects, as shown in Figure 2-12.

**FIGURE 2-12:** A 4 AA battery pack with a barrel jack adapter

Unplug the USB cable from your computer, insert four AA batteries into your battery pack, and plug your portable battery pack into your Arduino, as shown in Figure 2-13. If your batteries are charged, you can move your project around or embed it directly into a model stoplight!


**FIGURE 2-13:** Making the Stoplight portable by adding a battery pack

Now you'll level up this project. In the next section, we'll show you how to turn these LEDs into a model stoplight that you can mount in high-traffic areas of your house.

## BUILD THE STOPLIGHT ENCLOSURE

Once your Arduino isn't tethered to a computer, you can build any electronics project into a more permanent enclosure. The circuit on your breadboard is great, but you probably have to use your imagination to picture it as a stoplight. For maximum effect, the Stoplight just needs a good housing and lenses that will make the lights visible from a distance. The enclosure is optional if all you want to do is prototype, but we hope you'll try it out.

For this project, we'll show you how to build a more realistic-looking stoplight with some cardboard or cardstock, but you can use any material that you happen to have lying around. Be creative! Our example, shown in Figure 2-14, is made from some cardboard, ping-pong balls, and a bit of crafting skill.

**FIGURE 2-14:** An enclosure made from cardboard and ping-pong balls

**NOTE**

*If you're lucky enough to have access to a cutting machine like a Cricut, a Silhouette Cameo, or a laser cutter, these files should easily translate to those tools, too.*

You can either build a stoplight on your own using this project only as an inspiration or, if you want to reproduce this project exactly as you see it here, download the ZIP file of templates and sketches at *https://www.nostarch.com/arduinoinventor/*. Each project in this book includes templates that you can print, trace, and hand-cut the old-fashioned way with a craft knife and a metal ruler.

Extract the Project 2 files from the ZIP file, and print the Stoplight template PDF at full size if you'd like a cutting guide. With your templates in hand, collect the other items listed in "Other Materials and Tools" on page 39 and start building.

## Cardboard Construction

First, cut out the templates, shown in Figure 2-15. In our template, the housing body is a single piece of cardboard that is meant to be cut out, scored, and folded.

Trace the template onto your cardboard, and make careful note of the dashed lines, perhaps by drawing them on your cardboard in a different color. You'll score the cardboard along those lines to bend it, so whatever you do, don't cut along them yet.

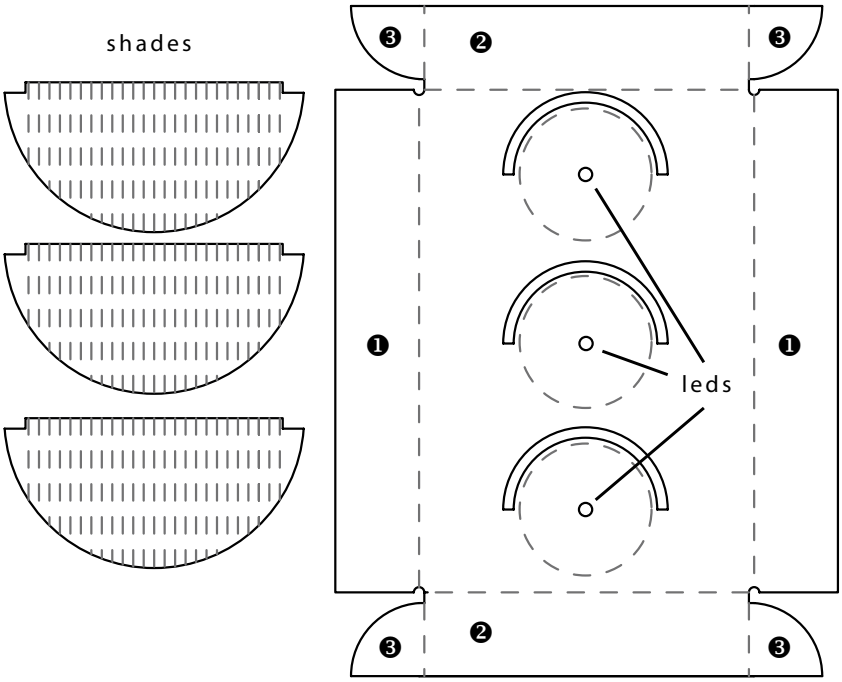



**FIGURE 2-15:**
Enclosure template for the Stoplight (not full size)

Once you have everything traced, cut out the stoplight pieces along the solid lines using a craft knife and a metal ruler, as shown in Figure 2-16. If you've never used a craft knife before, be sure to read "Using Craft Knives Safely" on page 56. Score the cardboard for the housing along each dotted line, on the exterior side of the cardboard. When scoring cardboard, you take a couple of shallow passes with the craft knife (don't cut all the way through). Don't score the shades yet.

**FIGURE 2-16:**
Scoring along the template with a craft knife and metal ruler

## USING CRAFT KNIVES SAFELY

You'll use craft knives a lot in this book, so it's important to know how to safely use them. Just like any tool, when used incorrectly, craft knives like the one here can cause injury.

Here are a few tips for using craft knives safely:

- Always pull the blade when slicing through sheet materials. Pushing or forcing the blade in any other direction raises the potential for slipping or breaking the blade.
- Be patient. Don't try to cut through the entire thickness of the material in a single pass. Make multiple passes with medium pressure. This will save your blade and also produce a cleaner finished product in the end.
- Use a straightedge made of metal, such as a metal ruler. If you use a wooden or plastic ruler as a straightedge, you run a higher chance of your blade catching the straightedge, rebounding off the material, and ultimately moving toward your hand.
- Keep your fingers out of the way. This may seem obvious, but accidents happen.
- If your knife starts to roll off your desk, let it fall, and just pick it up off the floor. If you reach for it and catch it before it falls, you run the risk of stabbing yourself in the hand. Ouch!
- Finally, use sharp, new, and intact blades. If a blade breaks, replace it. If a blade is dull, replace it. Cutting through paper and cardboard dulls blades very quickly. Keep a supply of extra blades around, and if it's starting to get hard to cut, replace the blade.

Once you have cut out your cardboard enclosure, add the mounting holes for the three LEDs; these should be at the little solid-lined circles inside the big dashed circles. One easy option is to carefully press a sharp pencil through the cardboard to make the holes. For cleaner holes, however, we suggest using a 3/16-inch drill bit and power drill to make holes in the cardboard, as in Figure 2-17. The LEDs are about 5 mm (about 0.197 inches) in diameter. You want the hole to be a nice, tight fit. So, a 3/16-inch hole (0.1875 inches) is perfect for making the fit snug for the LED.

Be careful when completing this step, and make sure to watch where your fingers and hands are relative to the drill bit. You don't want to drill into yourself! You can also use the drill bit without the drill and manually spin it through the cardboard if you don't have a drill or aren't comfortable using one.


**FIGURE 2-17:**
Drilling holes for the LEDs

Once you have the holes drilled, remove the three LEDs from your breadboard and insert them through the back side of the cardboard, as shown in Figure 2-18. Remember that standard traffic lights are usually ordered red, yellow, and green from the top to the bottom. Pay attention to where the LEDs connect on the board, because we're going to reconnect them at the end.

**FIGURE 2-18:** All three LEDs pressed into the cardboard

Next, bend the cardboard along the scored lines, as shown in Figure 2-19. Bend the vertical sides ❶ toward the interior, and then do the same with the top and bottom sides ❷ and the tabs ❸. (The sides and tabs are labeled in Figure 2-15.)

**FIGURE 2-19:** Prefolding the scored cardboard to form an enclosure for the Stoplight

Position the tabs ❸ inside the vertical sides ❶, and glue them in place as shown in Figure 2-20. You can use hot glue, tape, or craft glue—we prefer hot glue because it's easy to work with, sets quickly, and has a pretty strong bond.

Repeat this for the top and bottom corners. You should end up with a shallow rectangular box with an open back.

**FIGURE 2-20:**
Folding and gluing the cardboard housing

## Make the Stoplight Lenses

The Stoplight's lenses are made from ping-pong balls cut in half, but you can use anything that's moderately translucent.

If you're using ping-pong balls or something similar, carefully cut two balls in half. When doing this, place the ball against a cutting mat or thick piece of cardboard and hold it firmly at the sides with your fingertips. Carefully push the knife blade down toward the mat and into the ping-pong ball (making sure the blade isn't pointing at you or your hand) to make an incision as shown in Figure 2-21. Rotate the ping-pong ball and repeat until you've cut all the way through. Make sure to keep your fingers away from the blade, and always cut on a cutting mat or a piece of cardboard.

**FIGURE 2-21:**
Safely cutting a ping-pong ball

Once you have three ping-pong ball halves (you'll have four; one's an extra to use in future projects or as a small hat for your favorite stuffed animal), secure them with a dab of hot glue as shown in Figure 2-22.

**FIGURE 2-22:** The enclosure with ping-pong balls as lenses

## Make the Shades

Finally, add the shades to the Stoplight. For a nice curve, make a number of parallel scores, about 1/8 inch apart, as shown in Figure 2-23. There are example score lines in the template, so you can follow those. After making all of your scores, bend each shade into a curve, as shown in Figure 2-24.

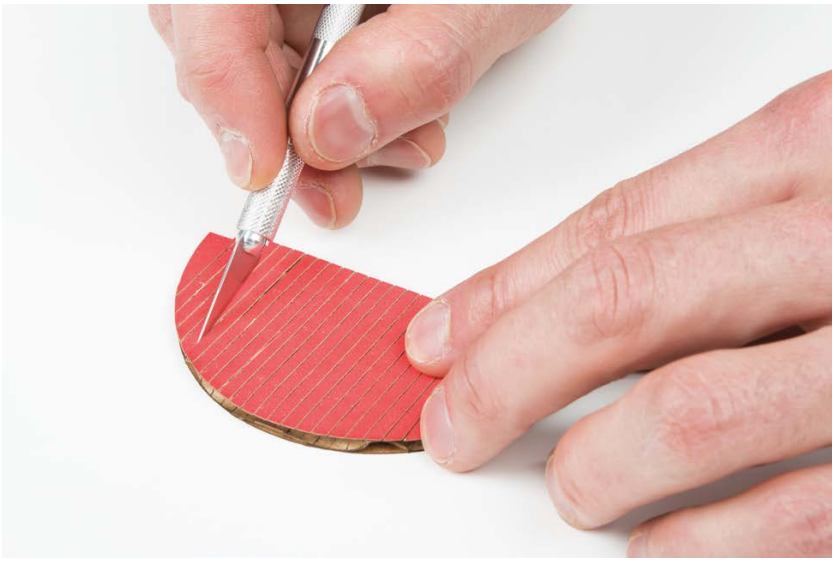


**FIGURE 2-23:** Scoring a shade

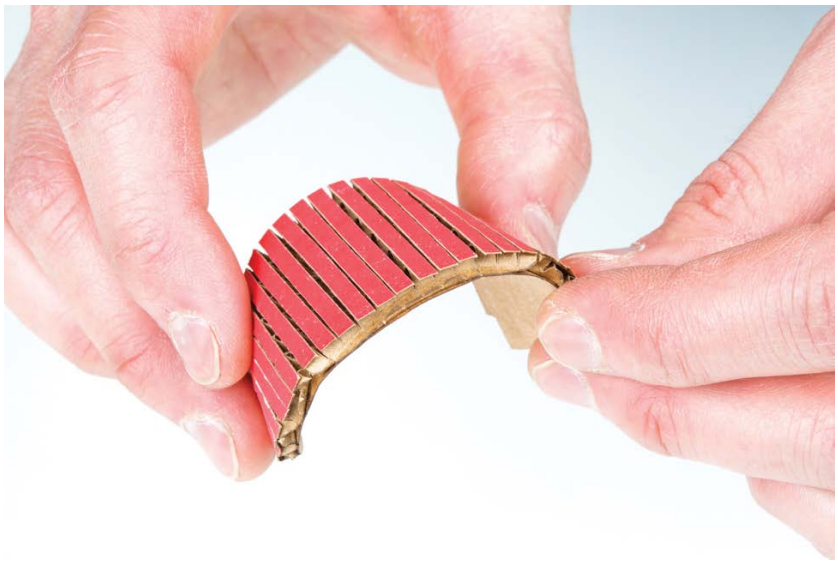


**FIGURE 2-24:**
Bending the shade into a curve

Once you have the shades bent and shaped to your liking, fit them into the housing just above each lens, as shown in Figure 2-25, and then glue them in place. If you're going for a more finished or realistic look, you can spray paint the housing black. Make sure you either remove the lenses or cover them with masking tape first so that they don't get coated in spray paint.

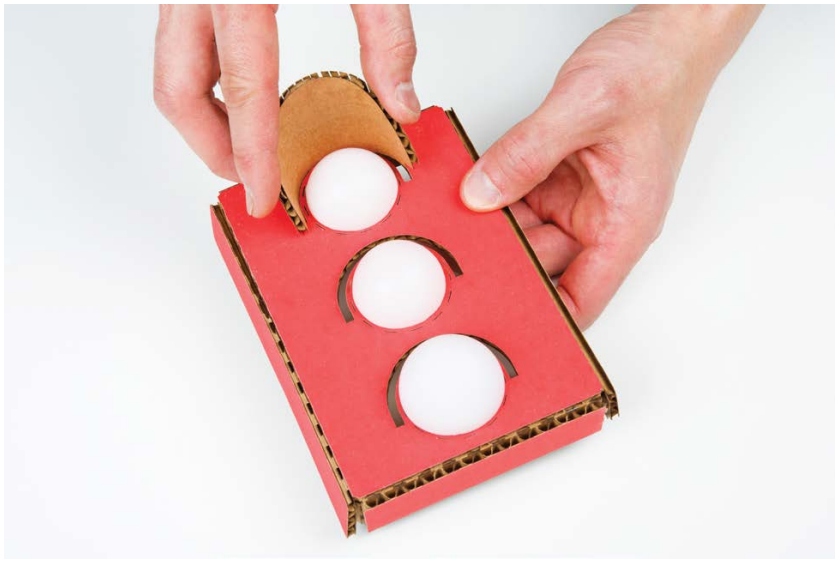


**FIGURE 2-25:**
Fitting a shade into the housing

## Mount the LEDs and Arduino

All you have left to do is to connect the LEDs from the new enclosure to your Arduino. First, use two male-to-female jumper wires (SparkFun PRT-09385) to extend each of the LEDs. You'll need a total of six of these jumper wires. Simply plug each LED leg into the

female end of the jumper wire. To keep things organized, we like to use black wires for the negative (shorter) leg and colored wires for the positive (longer) leg, as shown in Figure 2-26.

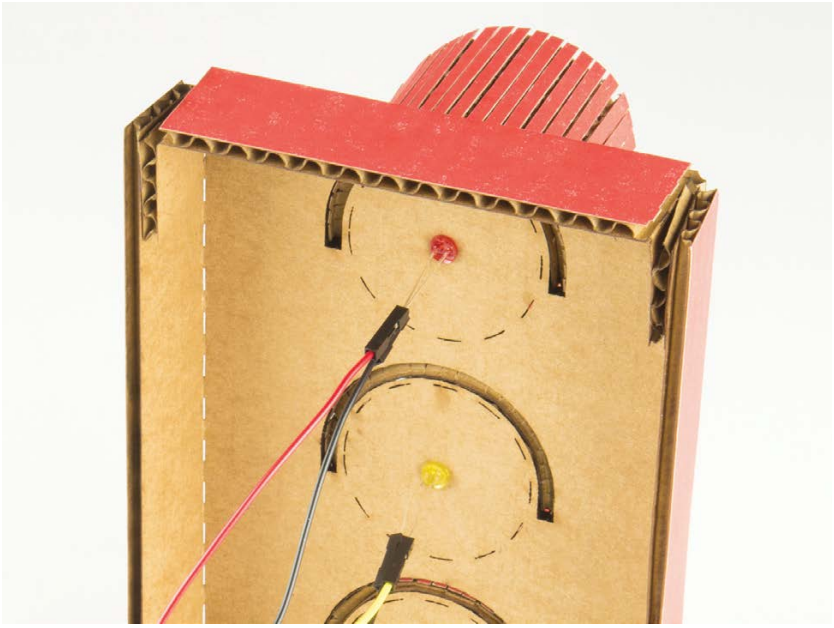


**FIGURE 2-26:** Attaching jumper wires to the LEDs

With the jumper wires connected to the LEDs, plug the male end into the breadboard in the same place where the LED came out, as shown in Figure 2-27. Again, pay attention to which LED goes where. If you don't remember, consult the original diagram in Figure 2-10.

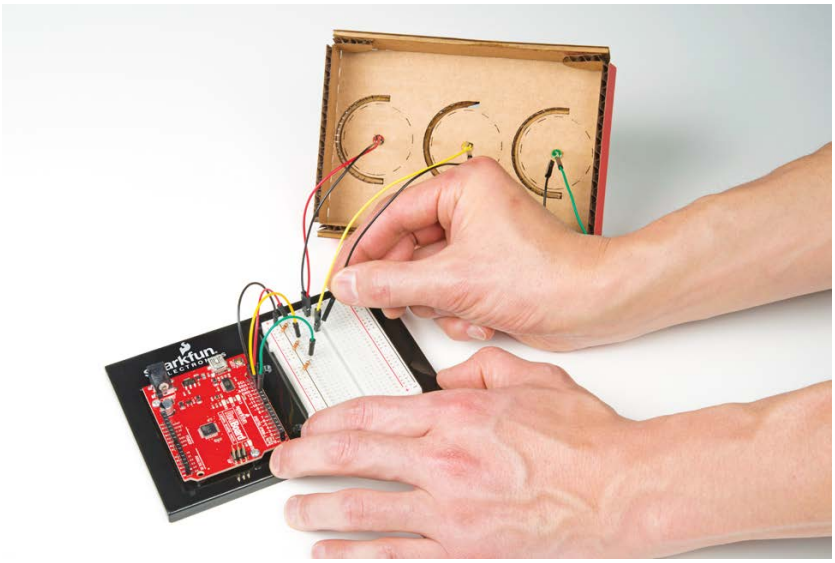


**FIGURE 2-27:** Inserting the male end of each jumper wire into the breadboard

Check to make sure your connections work by plugging in the Arduino to your computer or to a battery pack. If one of the lights isn't working, try jiggling the connections or double-checking that the wires are plugged into the correct row on the breadboard.

You can either leave the Arduino and breadboard outside the Stoplight housing or tack them inside the housing with glue or double-sided tape. Whatever you decide, when you're done, power up your Stoplight, and go find a busy hallway intersection in need of traffic safety.

Figure 2-28 shows the finished Stoplight in all its glory.

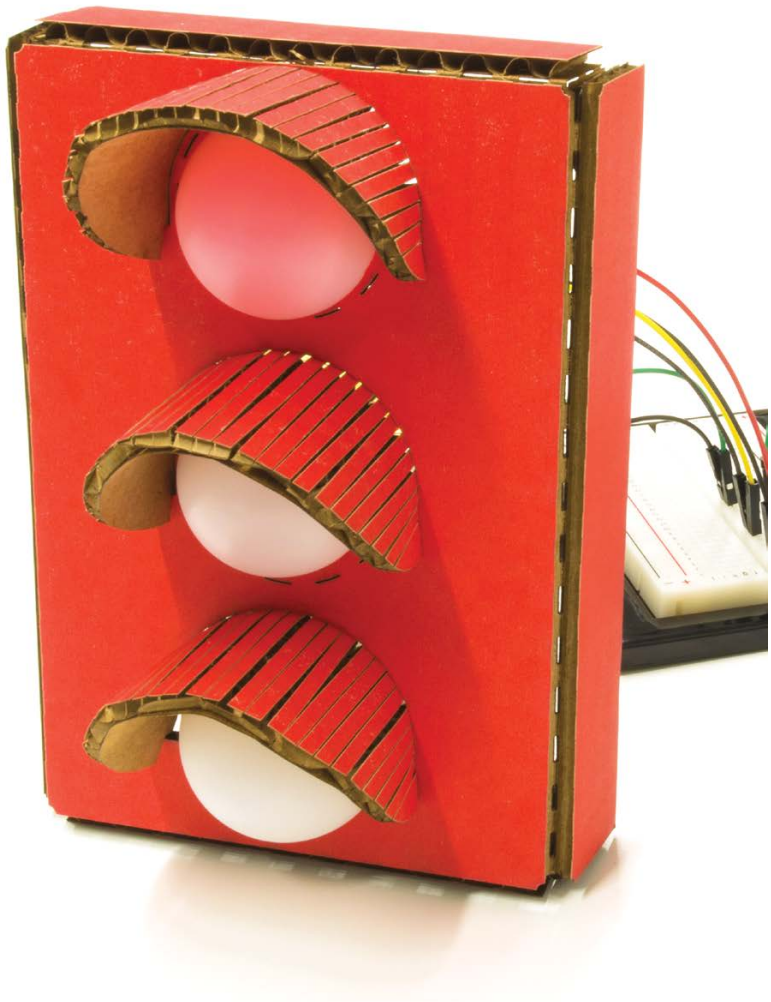


**FIGURE 2-28:**
Finished Stoplight project

## GOING FURTHER

The concepts you saw while building the Stoplight, such as timing the control of output (LEDs), can be applied to a number of different uses in your house and life. Here are a couple of suggestions for adapting the Stoplight.

### Hack

The basic concept of a stoplight is all about timing. When else would a timer be useful? What about changing the code to help you time frying an egg? You could rework the Stoplight so that the red LED is lit while the egg is still in a state of "iffy" or "rare" doneness, the yellow LED lights when it's almost cooked the way you like it, and then the green LED lights when the egg is done.

We can't give you the timing, as we probably have different preferences for how we like our eggs cooked. There are also a number of variables that will affect the timing, like the temperature, the type of pan, and the size of the egg. You'll have to figure that all out on your own.

In the code, you'd need to work with pretty big numbers for the delay, since it's measured in milliseconds. To set a delay in minutes, all you need is a little multiplication. Remember that 1,000 ms equals 1 second; multiply by 60, and you'll find that 60,000 ms equals 60 seconds, or 1 minute. For a delay of 3 minutes, you can multiply 3 by 60,000 directly in the `delay()` function, like this:

```
delay(60,000 * 3);
```

You may be wondering how long you can set the `delay()` function for. The data type that `delay()` receives is an *unsigned long*, which is any number that falls in the range of 0 to 4,294,967,295. So the maximum delay is 1,193 hours or so. Pretty cool! Knowing this, is there anything else you'd want to time with the `delay()` function?

## Modify

If you're looking to make this project more permanent and sturdy, you can solder wires to the LEDs instead of using the male-to-female jumpers. If you've never soldered before, turn to "How to Solder" on page 302 for some soldering instructions before you start. You'll need to snip the end off of a male-to-male jumper wire, strip the insulation back about 1/2 inch using wire strippers, and then solder the stripped end to each leg of a trimmed LED, as shown in Figure 2-29. Notice that we twisted the wire around the leg of the LED to hold it securely while soldering. After soldering, the connection will be more durable, and you'll be able to use the LEDs for other projects since the other end is still a male jumper.

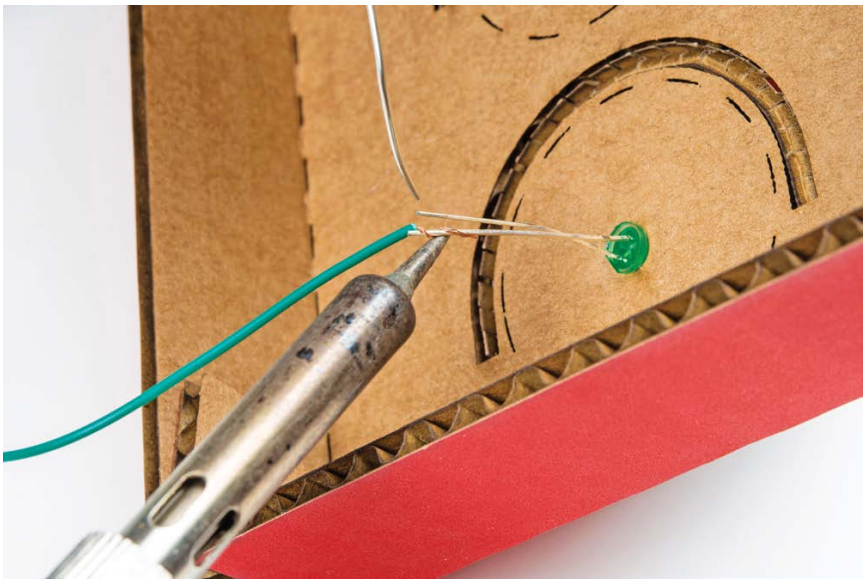


**FIGURE 2-29:** Soldering a cut jumper wire to an LED

Though this project looks impressive, the programming and hardware are pretty simple. As you read about sensors and logic over the next few chapters, we encourage you to think back to this project and brainstorm ways you can elaborate on it with what you learn.

# LINUX BASICS FOR HACKERS

GETTING STARTED WITH *NETWORKING*, *SCRIPTING*, AND *SECURITY* IN KALI

OCCUPYTHEWEB

# 3

# ANALYZING AND MANAGING NETWORKS

Understanding networking is crucial for any aspiring hacker. In many situations, you'll be hacking something over a network, and a good hacker needs to know how to connect to and interact with that network. For example, you may need to connect to a computer with your Internet Protocol (IP) address hidden from view, or you may need to redirect a target's Domain Name System (DNS) queries to your system; these kinds of tasks are relatively simple but require a little Linux network know-how. This chapter shows you some essential Linux tools for analyzing and managing networks during your network-hacking adventures.

## Analyzing Networks with ifconfig

The `ifconfig` command is one of the most basic tools for examining and interacting with active network interfaces. You can use it to query your active network connections by simply entering **`ifconfig`** in the terminal. Try it yourself, and you should see output similar to Listing 3-1.

```
kali > ifconfig
❶eth0Linkencap:EthernetHWaddr 00:0c:29:ba:82:0f
❷inet addr:192:168.181.131 ❸Bcast:192.168.181.255 ❹Mask:255.255.255.0
--snip--
❺lo Linkencap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
--snip--
❻wlan0 Link encap:EthernetHWaddr 00:c0:ca:3f:ee:02
```

*Listing 3-1: Using ifconfig to get network information*

As you can see, the command `ifconfig` shows some useful information about the active network interfaces on the system. At the top of the output is the name of the first detected interface, `eth0` ❶, which is short for Ethernet0 (as you'll recall, Linux starts counting at 0 rather than 1). This is the first wired network connection. If there were more wired Ethernet interfaces, they would show up in the output using the same format (`eth1`, `eth2`, and so on).

The type of network being used (`Ethernet`) is listed next, followed by `HWaddr` and an address; this is the globally unique address stamped on every piece of network hardware—in this case, the network interface card (NIC), usually referred to as the media access control (MAC) address.

The second line contains information on the IP address currently assigned to that network interface (in this case, 192.168.181.131 ❷); the `Bcast` ❸, or *broadcast address*, which is the address used to send out information to all IPs on the subnet; and finally the *network mask* (`Mask` ❹), which is used to determine what part of the IP address is connected to the local network. You'll also find more technical info in this section of the output, but it's beyond the scope of this Linux networking basics chapter.

The next section of the output shows another network connection called `lo` ❺, which is short for *loopback address*, and sometimes called *localhost*. This is a special software address that connects you to your own system. Software and services not running on your system can't use it. You would use `lo` to test something on your system, such as your own web server. The localhost is generally represented with the IP address 127.0.0.1.

The third connection is the interface `wlan0` ❻. This only appears if you have a wireless interface or adapter, like I clearly do here. Note that it also displays the MAC address of that device (`HWaddr`).

This information from `ifconfig` enables you to connect to and manipulate your local area network (LAN) settings, an essential skill for anyone who wants to learn how to hack.

## Checking Wireless Network Devices with iwconfig

If you have a wireless adapter, you can use the `iwconfig` command to gather information such as the adapter's IP address, its MAC address, what mode it's in, and other crucial information you can use in wireless hacking. The information you can glean from this command is particularly important when you're using wireless hacking tools like `aircrack-ng`.

Using the terminal, let's take a look at some wireless devices with `iwconfig` (see Listing 3-2).

```
kali > iwconfig
wlan0 IEEE 802.11bg ESSID:off/any
Mode:Managed Access Point: Not Associated Tx-Power=20 dBm
*
*
*
lo   no wireless extensions

eth0  no wireless extensions
```

*Listing 3-2: Using `iwconfig` to get information on wireless adapters*

The output here tells us that the only network interface with wireless extensions, as we would expect, is `wlan0`. Neither `lo` nor `eth0` have any wireless extensions.

For `wlan0`, we learn what 802.11 IEEE wireless standards our device is capable of: `bg`, two early wireless communication standards. Most wireless devices now include both `bg` and `n` (`n` is the latest standard).

We also learn from `iwconfig` the mode of the wireless extension (in this case, `Mode:Managed`, in contrast to monitor or promiscuous mode). We'll need promiscuous mode for cracking wireless passwords.

Next, we can see that the wireless adapter is not connected (`Not Associated`) to an access point (AP), and that its power is 20dBm, which represents the strength of signal. We'll spend more time with this information in Chapter 14.

## Changing Your Network Information

Being able to change your IP address and other network information is a useful skill because it helps hackers access other networks and appear as a trusted device on those networks. For example, in a denial-of-service (DoS) attack, you can spoof your IP so that that the attack appears to come from another source, thus helping you evade IP capture during forensic analysis. This is a relatively simple task in Linux, and it's done with the `ifconfig` command.

### *Changing Your IP Address*

To change your IP address, enter **`ifconfig`** followed by the interface you want to reassign and the new IP address you want assigned to that interface. For example, to assign the IP address 192.168.181.115 to interface `eth0`, you would enter the following:

```
kali > ifconfig eth0 192.168.181.115
kali >
```

When you do this correctly, Linux will simply return the command prompt and say nothing. This is a good thing!

Then, when you again check your network connections with `ifconfig`, you should see that your IP address has changed to the new IP address you just assigned.

### Changing Your Network Mask and Broadcast Address

You can also change your network mask (netmask) and broadcast address with the `ifconfig` command. For instance, if you want to assign that same `eth0` interface with a netmask of 255.255.0.0 and a broadcast address of 192.168.1.255, you would simply enter the following:

```
kali > ifconfig eth0 192.168.181.115 netmask 255.255.0.0 broadcast 192.168.1.255
kali >
```

Once again, if you've done everything correctly, Linux just responds with a new command prompt. Now enter **`ifconfig`** again to verify that each of the parameters has been changed accordingly.

### Spoofing Your MAC Address

You can also use `ifconfig` to change your MAC address (or `HWaddr`). Remember that the MAC address is globally unique and is often used as a security measure to keep hackers out of networks—or used to trace them. Changing your MAC address to spoof a different MAC address is almost trivial and neutralizes those security measures. It's a very useful technique for bypassing network access controls.

To spoof your MAC address, simply use the `ifconfig` command's `down` option to take down the interface (`eth0` in this case). Then enter the `ifconfig` command followed by the interface name (`hw` for hardware, `ether` for Ethernet) and the new spoofed MAC address. Finally, bring the interface back up with the `up` option for the change to take place. Here's an example:

```
kali > ifconfig eth0 down
kali > ifconfig eth0 hw ether 00:11:22:33:44:55
kali > ifconfig eth0 up
```

Now, when you go back and check your settings with `ifconfig`, you should see that `HWaddr` has changed to your new spoofed IP address!

### Assigning New IP Addresses from the DHCP Server

Linux has a Dynamic Host Configuration Protocol (DHCP) server that runs a *daemon*—a process that runs in the background—called `dhcpd`, or the *dhcp daemon*. The DHCP server assigns IP addresses to all the systems on the subnet and keeps log files of which IP address is allocated to which machine

at any one time. This makes it a great resource for forensic analysts to trace hackers with after an attack. For that reason, it's useful to understand how the DHCP server works.

Usually, to connect to the internet from a LAN, you must have a DHCP-assigned IP. Therefore, after setting a static IP address, you must return and get a new DHCP-assigned IP address. You can always reboot your system to get a new DHCP-assigned IP address, but I'll show you how to retrieve a new DHCP without having to shut your system down and restart it.

To request an IP address from DHCP, simply call the DHCP server with the command `dhclient` followed by the interface you want the address assigned to. Different Linux distributions use different DHCP clients, but Kali is built on Debian, which uses `dhclient`; therefore, you can assign a new address like this:

```
kali > dhclient eth0
```

The `dhclient` command sends a `DHCPDISCOVER` request from the network interface specified (here, `eth0`). It then receives an offer (`DHCPOFFER`) from the DHCP server (192.168.181.131 in this case) and confirms the IP assignment to the DHCP server with a dhcp request.

```
kali > ifconfig
eth0Linkencap:EthernetHWaddr 00:0c:29:ba:82:0f
inet addr:192:168.181.131 Bcast:192.168.181.131 Mask:255.255.255.0
```

Depending on the configuration of the DHCP server, the IP address assigned in each case might be different.

Now when you enter `ifconfig`, you should see that the DHCP server has assigned a new IP address, a new broadcast address, and new netmask to your network interface `eth0`.

## Manipulating the Domain Name System

Hackers can find a treasure trove of information on a target in its Domain Name System (DNS). DNS is a critical component of the internet, and although it's designed to translate domain names to IP addresses, a hacker can use it to garner information on the target.

### *Examining DNS with dig*

DNS is the service that translates a domain name like *hackers-arise.com* to the appropriate IP address; that way, your system knows how to get to it. Without DNS, we would all have to remember thousands of IP addresses for our favorite websites, which is no small task, even for a savant.

One of the most useful commands for the aspiring hacker is `dig`, which offers a way to gather DNS information about a target domain. The stored DNS information can be a key piece of early reconnaissance to obtain

before attacking. This information could include the IP address of the target's nameserver (the server that translates the target's name to an IP address), the target's email server, and potentially any subdomains and IP addresses.

For instance, enter **dig hackers-arise.com** and add the **ns** option (short for *nameserver*). The nameserver for *hackers-arise.com* is displayed in the ANSWER SECTION of Listing 3-3.

```
kali > dig hackers-arise.com ns
*
*
*
;; QUESTION SECTION:
;hackers-arise.com.    IN   NS

;;ANSWER SECTION:
hackers-arise.com.  5  IN   NS   ns7.wixdns.net.
hackers-arise.com.  5  IN   NS   ns6.wixdns.net.

;; ADDITIONAL SECTION:
ns6.wixdns.net.   5  IN   A   216.239.32.100
*
*
*
```

*Listing 3-3: Using dig and its ns option to get information on a domain nameserver*

Also note in the ADDITIONAL SECTION that this dig query reveals the IP address (216.239.32.100) of the DNS server serving *hackers-arise.com*.

You can also use the dig command to get information on email servers connected to a domain by adding the mx option (mx is short for *mail exchange server*). This information is critical for attacks on email systems. For example, info on *www.hackers-arise.com* email servers is shown in the AUTHORITY SECTION of Listing 3-4.

```
kali > dig hackers-arise.com mx
*
*
*
;; QUESTION SECTION:
;hackers-arise.com.    IN   MX

;;AUTHORITY SECTION:
hackers-arise.com.  5  IN   SOA   ns6.wixdns.net. support.wix.com 2016052216
10800 3600 604 800 3600

*
*
*
```

*Listing 3-4: Using dig and its mx option to get information on a domain mail exchange server*

The most common Linux DNS server is the Berkeley Internet Name Domain (BIND). In some cases, Linux users will often refer to DNS as BIND, but don't be confused: DNS and BIND both simply map individual domain names to IP addresses.

### Changing Your DNS Server

In some cases, you may want to use another DNS server. To do so, you simply edit a plaintext file named */etc/resolv.conf* on the system. Open that file in a text editor—I'm using Leafpad. Then, on your command line, enter the precise name of your editor followed by the location of the file and the filename. For example,

```
kali > leafpad /etc/resolv.conf
```

will open the *resolv.conf* file in the */etc* directory in my specified graphical text editor, Leafpad. The file should look something like Figure 3-1.


*Figure 3-1: A typical* resolv.conf *file in a text editor*

As you can see on line 3, my nameserver is set to a local DNS server at 192.168.181.2. That works fine, but if I want to add or replace that DNS server with, say, Google's public DNS server at 8.8.8.8, I'd add the following line in the */etc/resolv.conf* file to specify the nameserver:

```
nameserver  8.8.8.8
```

Then I would just need to save the file. However, you can also achieve the same result exclusively from the command line by entering the following:

```
kali > echo "nameserver 8.8.8.8"> /etc/resolv.conf
```

This command echoes the string `nameserver 8.8.8.8` and redirects it (>) to the file `/etc/resolv.conf`, replacing the current content. Your */etc/resolv.conf* file should now look like Figure 3-2.


*Figure 3-2: Changing the* resolv.conf *file to specify Google's DNS server*

If you open the */etc/resolv.conf* file now, you should see that it points the DNS requests to Google's DNS server rather than your local DNS server. Your system will now go out to the Google public DNS server to resolve domain names to IP addresses. This can mean domain names take a little longer to resolve (probably milliseconds). Therefore, to maintain speed but still keep the option of using a public server, you might want to retain the local DNS server in the *resolv.conf* file and follow it with a public DNS server. The operating system queries each DNS server listed in the order they appear in */etc/resolv.conf*, so the system will only refer to the public DNS server if the domain name can't be found in the local DNS server.

**NOTE** *If you're using a DHCP address and the DHCP server provides a DNS setting the DHCP server will replace the contents of the file when it renews the DHCP address.*

### Mapping Your Own IP Addresses

A special file on your system called the *hosts* file also performs domain name–IP address translation. The *hosts* file is located at */etc/hosts*, and kind of like DNS, you can use it to specify your own IP address–domain name mapping. In other words, you can determine which IP address your browser goes to when you enter *www.microsoft.com* (or any other domain) into the browser, rather than let the DNS server decide. As a hacker, this can be useful for hijacking a TCP connection on your local area network to direct traffic to a malicious web server with a tool such as `dnsspoof`.

From the command line, type in the following command (you can substitute your preferred text editor for `leafpad`):

```
kali> leafpad /etc/hosts
```

You should now see your *hosts* file, which will look something like Figure 3-3.

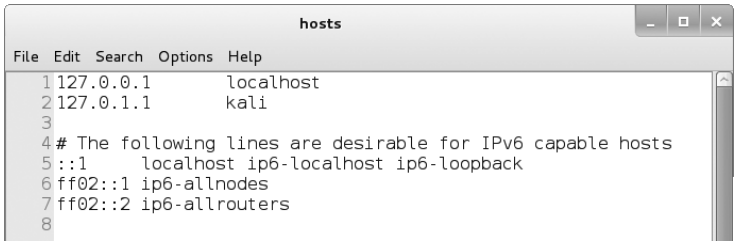



*Figure 3-3: A default Kali Linux hosts file*

By default, the *hosts* file only contains a mapping for your localhost, at 127.0.0.1, and your system's hostname (in this case, Kali, at 127.0.1.1). But you can add any IP address mapped to any domain you'd like. As an example of how this might be used, you could map *www.bankofamerica.com* to your local website, at 192.168.181.131.

```
127.0.0.1       localhost
127.0.0.1       kali.kali 2016  kali
192.168.181.131 bankofamerica.com

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Make certain you press TAB between the IP address and the domain key—not the spacebar.

As you get more involved in your hacking endeavors and learn about tools like `dnsspoof` and Ettercap, you'll be able to use the *hosts* file to direct any traffic on your LAN that visits *www.bankofamerica.com* to your web server at 192.168.181.131.

Simple, right?

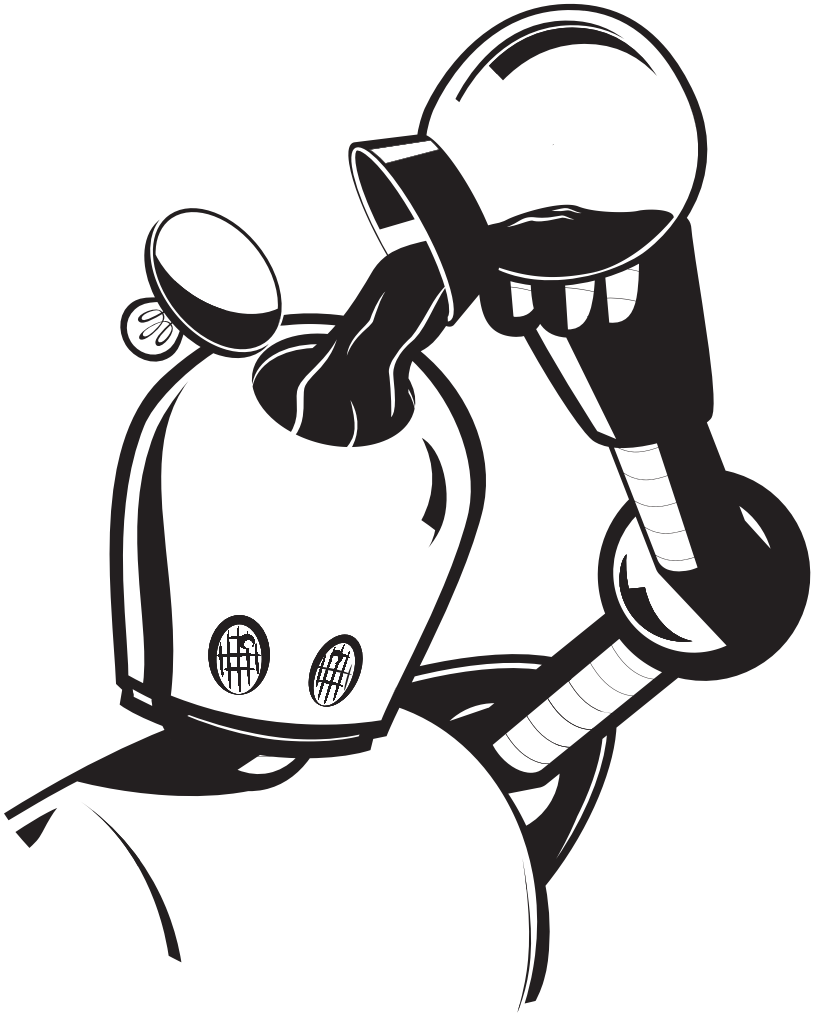
## Summary

Any hacker needs some basic Linux networking skills to connect, analyze, and manage networks. As you progress, these skills become more and more useful for reconnaissance, spoofing, and connecting to target systems.

## Exercises

Before you move on to Chapter 4, try out the skills you learned from this chapter by completing the following exercises:

1. Find information on your active network interfaces.
2. Change the IP address on `eth0` to 192.168.1.1.
3. Change your hardware address on `eth0`.
4. Check whether you have any available wireless interfaces active.
5. Reset your IP address to a DHCP-assigned address.
6. Find the nameserver and email server of your favorite website.
7. Add Google's DNS server to your */etc/resolv.conf* file so your system refers to that server when it can't resolve a domain name query with your local DNS server.



Founded in 1994, No Starch Press is one of the few remaining independent technical book publishers. We publish the finest in geek entertainment—unique books on technology, with a focus on open source, security, hacking, programming, alternative operating systems, and LEGO. Our titles have personality, our authors are passionate, and our books tackle topics that people care about.

**VISIT WWW.NOSTARCH.COM FOR A COMPLETE CATALOG.**

NO STARCH PRESS 2018 CATALOG FOR HUMBLE BOOK BUNDLE: MAKERSPACE. COPYRIGHT © 2018 NO STARCH PRESS, INC. ALL RIGHTS RESERVED. 20 EASY RASPBERRY PI PROJECTS © RUI SANTOS AND SARA SANTOS. THE LEGO ARCHITECTURE IDEA BOOK © ALICE FINCH. THE RUST PROGRAMMING LANGUAGE © STEVE KLABNIK AND CAROL NICHOLS, WITH CONTRIBUTIONS FROM THE RUST COMMUNITY. CODING WITH MINECRAFT © AL SWEIGART. CRACKING CODES WITH PYTHON © AL SWEIGART. PRACTICAL SQL © ANTHONY DEBARROS. SERIOUS CRYPTOGRAPHY © JEAN-PHILIPPE AUMASSON. ARDUINO INVENTOR'S GUIDE © BRIAN HUANG AND DEREK RUNBERG. LINUX BASICS FOR HACKERS © OCCUPYTHEWEB. NO STARCH PRESS AND THE NO STARCH PRESS LOGO ARE REGISTERED TRADEMARKS OF NO STARCH PRESS, INC. NO PART OF THIS WORK MAY BE REPRODUCED OR TRANSMITTED IN ANY FORM OR BY ANY MEANS, ELECTRONIC OR MECHANICAL, INCLUDING PHOTOCOPYING, RECORDING, OR BY ANY INFORMATION STORAGE OR RETRIEVAL SYSTEM, WITHOUT THE PRIOR WRITTEN PERMISSION OF NO STARCH PRESS, INC.

no starch press