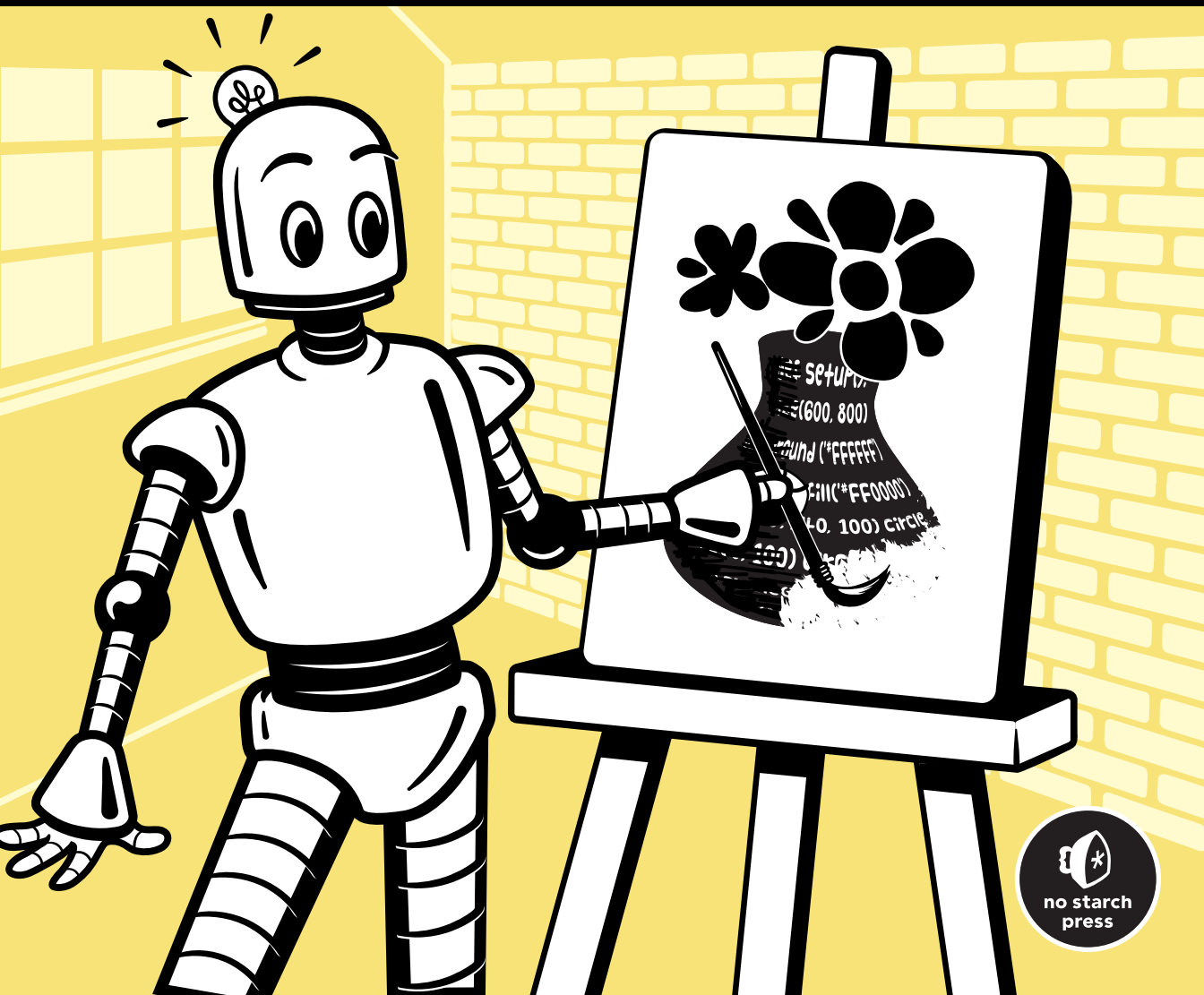# LEARN PYTHON VISUALLY

CREATIVE CODING WITH PROCESSING.PY

TRISTAN BUNN

LEARN PYTHON VISUALLY

LEARN PYTHON VISUALLY

CREATIVE CODING WITH PROCESSING.PY

by Tristan Bunn

no starch
press

San Francisco

About the Author

Tristan Bunn kicked off his web design career back in the days of PlayStation 1, grunge music, and dial-up modems. Since then, he's worked on a diverse range of digital projects for varied clients. He's currently involved in lecturing, research, and work that blends code, interaction, interface design, and creativity. Tristan has years of experience teaching coding for art, games, web, and other creative technologies.

About the Tech Reviewer

Paddy Gaunt studied engineering at Cambridge University (UK), working in the chemical and gas industries as well as textile manufacturing. Much of the time, he had the responsibility of implementing IT systems as these became a more significant part of management and marketing. Since its launch in 2012, he has been the chief maintainer of the pi3d Python module for fast 3D graphics on the Raspberry Pi microcomputer.

BRIEF CONTENTS

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

When I first encountered programming code, I stared, amazed and bewildered, at a screen of obscure commands and symbols and wondered how anybody could understand it, let alone write it. I'd hit the F5 key, and the program would magically spawn a cityscape in which two players, depicted as gorillas, could hurl explosive bananas at each other. I tried changing a few lines to see what would happen, and on occasion, it was something predictable or cool. More often than not, the game would simply fail to run. In a futile attempt to be helpful, the computer would diagnose my errors, rambling on about syntax and "illegal" operations of varying description.

For some years thereafter, I was content to avoid learning to program. That began to change when I became interested in making my creative work more interactive. You may already have encountered a few of the same

barriers that frustrated me. Maybe you were getting by just fine with visual tools but then hit a wall. Or to your disappointment (and horror?), you discovered that what you sought to accomplish required delving into code.

Software applications, with all of their graphical widgets, make us feel like we're in control. The illusion, however, soon fades when you discover that the tool you desire is missing. Through learning to program, you gain a true mastery of your computer.

Who Is This Book For?

This book assumes no prior programming experience. It strives to make the process of learning to program as visual and entertaining as possible. The content is based on my extensive experience teaching first-time coders, designers, and interactive media students. The skills and knowledge you'll gain are fundamental to programming for an ever-expanding horizon of creative technologies, such as games, the web, augmented/virtual reality, and even visual effects for films.

If you're an artist, student, designer, researcher, or just somebody keen on picking up coding skills, Python Mode for Processing is excellent for learning to program in a visual context.

For anybody with prior programming experience, this book would be useful for learning Python, Python Mode for Processing, or creative coding techniques.

You may have experience with another *visual* programming language— something like Scratch, where you connect together graphical elements like boxes, icons, and arrows. Python is not such a language—rather, it is a *textual* programming language that requires you to type code. To make learning visual, though, you'll focus on writing code that produces drawings, patterns, animations, data visualizations, user interfaces, and simulations. This approach not only makes for cool-looking graphics, but also helps you visualize the underlying concepts of programming.

What Is Python Mode for Processing?

Python Mode for Processing combines the *Python* programming language and *Processing*, a development environment for interactive and graphics programming. You'll also see Python Mode for Processing referred to as *Processing.py*. The project started as a command line tool named Processing.py, but its developer decided to label it Python Mode when it was made available for the Processing development environment. In this book, you can consider the terms largely interchangeable.

Python is one of the most popular programming languages in use today. There are many good reasons for this, but here's why *you* should care. First, Python is a beginner-friendly language. It's more approachable than languages like Java or C++, so you'll find it easier to read, write, and

understand. Second, it's a general-purpose language, suitable for programming artificial intelligence (AI), games, simulations, web applications, and just about everything in between.

Processing, which has been around since the early 2000s, is composed of a programming language and an editor for writing and compiling code. It provides a collection of special commands that allow you to draw, animate, and handle user input by using code. The creators, Casey Reas and Ben Fry, developed Processing to make programming more accessible for designers and artists, although its thriving user base has grown to include researchers, hobbyists, and educators.

Java is the basis for the original Processing programming language, but other variants have since appeared, including JavaScript (*p5.js*) and Ruby (*JRubyArt*) versions. In 2010, Jonathan Feinberg created Processing.py, which you can think of as a sort of extension for Processing that allows you to write Python instead of Java-esque code.

Both Python and Processing are open source and won't cost you a cent. What's more, you can use them on just about any platform, including Linux, macOS, and Microsoft Windows.

What Are Algorithms?

You'll encounter the term *algorithm* frequently in the domain of programming. You can think of an algorithm as a set of rules a computer or machine must follow to achieve a particular goal. As an example, an algorithm for making a cup of instant coffee would read as follows:

1. Place one teaspoon of coffee granules in a mug.
2. Fill the kettle with water.
3. Switch on the kettle.
4. Once the water has boiled, add 240 ml boiling water to the mug.
5. Add one level teaspoon of sugar to the same mug.
6. Stir the contents.
7. Serve.

However, this set of steps is insufficient for programming a real-life coffee-making robot. Should the sizes of the mugs vary, smaller ones would overflow. Furthermore, the robot would ignore any requests for milk or extra sugar. Computers cannot make any assumptions, and require explicit and unambiguous direction, communicated in a language that machines understand—like Python. Learning the Python language may be the hurdle you face initially, but as you grow more fluent, the challenge will shift toward the mastery of algorithmic thinking.

What Is Creative Coding?

Creative coding is computer programming for creative output. This broad term encompasses, but is not limited to, computer-generated audio and visual art, interactive installations, experimental games, and data visualizations.

Take, for example, Frederic Brodbeck's *Cinemetrics* project. Using Python, Brodbeck developed a program that analyzes DVD movie data to generate visual *fingerprints* of films. The fingerprint is an open ring formed from many segments; a single segment represents a span of 10 shots, and the concentric bands show the color breakdown for each of those segments. The diagonal length of each segment indicates the amount of motion. Figure 1 is a *Cinemetrics* fingerprint for the film *Quantum of Solace* (2008).
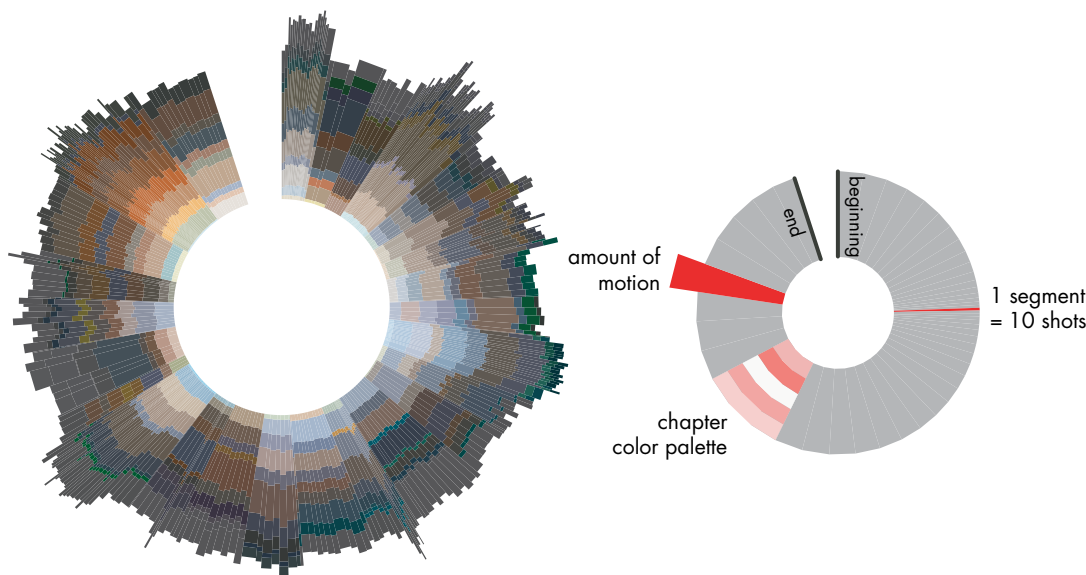
Figure 1: Quantum of Solace *fingerprint, created by Frederic Brodbeck. Screenshot from* http://cinemetrics.site/.

The fingerprints can also be animated, in which case motion is instead visualized using pulsating segments. An interactive interface provides a selection of presets and filters so that you can arrange fingerprints alongside one another and make comparisons—for example, between originals and remakes, different genres, the works of a single director, and so forth. Figure 2 compares (from left to right) *2001: A Space Odyssey* (1968), *The Simpsons Movie* (2007), and a soccer match.
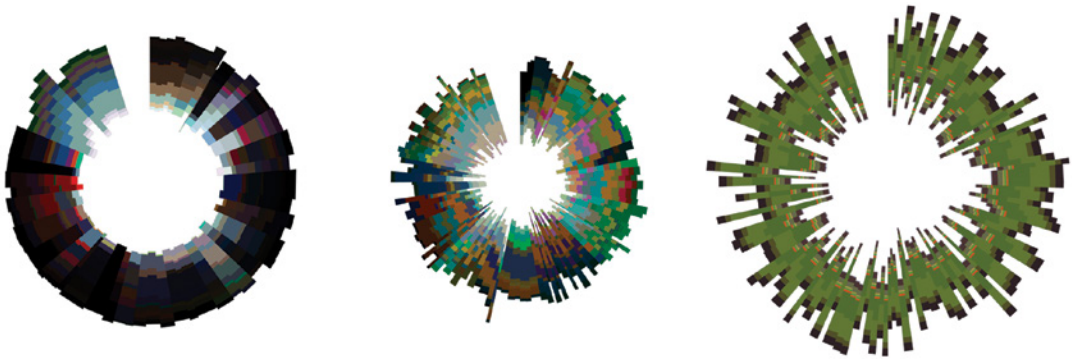
Figure 2: Fingerprints comparing (left to right) 2001: A Space Odyssey, The Simpsons Movie, *and a soccer match. Screenshots from* http://cinemetrics.site/.

Many creative coding projects employ a similar underlying approach, in which data is fed into a program to influence how it controls output. Music visualizations with real-time audio synchronization—like those featured in popular media player software—are a good example. However, you can experiment with plenty of other data sources, such as web feeds, fitness trackers, environmental sensors, and a plethora of public datasets.

In some instances, it's useful to opt for randomized data values. Consider procedurally generated game content. As opposed to constructing levels manually, you can program games to generate dungeon layouts, terrain, narrative elements, and enemy spawn locations automatically. Of course, such games should include sensible constraints; for instance, a cap on the total number of enemies that can appear at once, and algorithms for ensuring that stage layouts are not impossible to traverse.

Game characters may be composed using a random selection of modular components, or generated entirely from shapes and formulas. As an example, I've written a Processing Python program that generates the randomized microbial beasties displayed in Figure 3. The code—an adaptation of Lieven Menschaert's NodeBox script *Aquatics!*—spawns a creature with a random fill color, shape (defined by something named the *superformula*), and no fewer than three eyes. There's a 70 percent chance that hair will grow along the creature's edges, which can be swayed by the force of a randomly directed current.
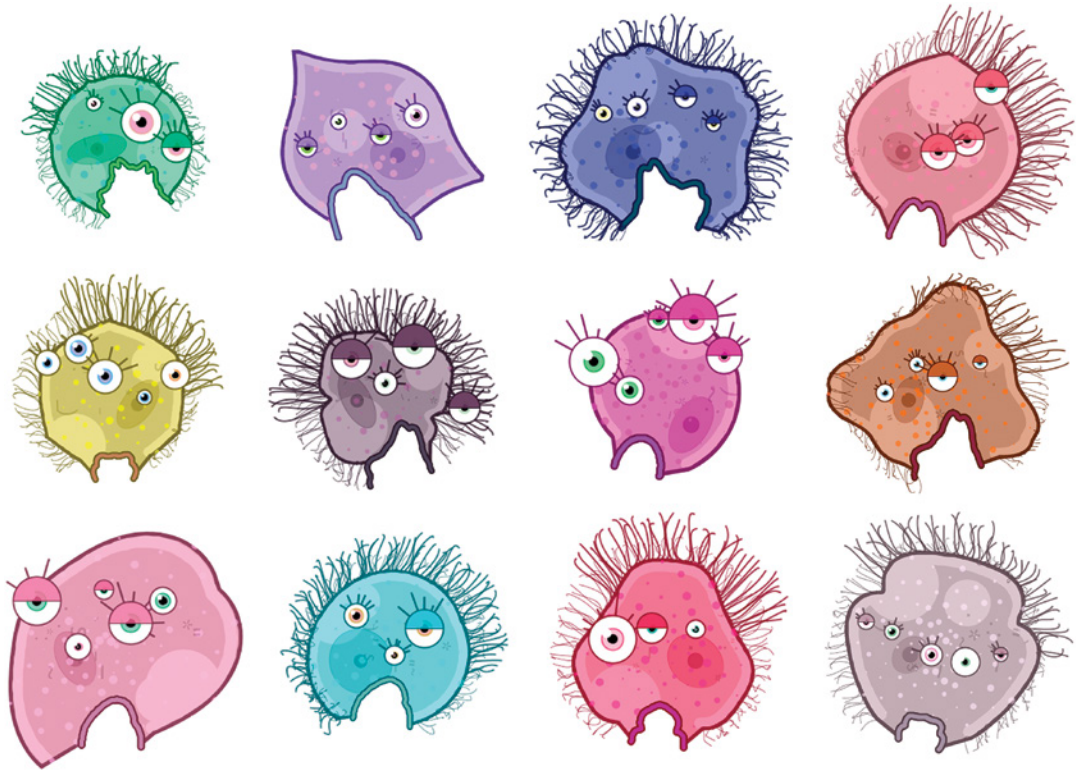
Figure 3: A Processing.py adaption of Lieven Menschaert's NodeBox script Aquatics!

Countless examples of cool, creative coding projects exist—from robots that doodle and write poetry, to evolutionary simulators, and even a program that pores over satellite imagery in search of architecture or infrastructure that resembles letters (*The Aerial Bold Project* by Benedikt Groß and Joey Lee, 2016).

Perhaps this creative coding thing sounds a bit too artsy for you? Processing also isn't ideal for that race car sim you've always dreamed of building, and it's definitely no good for backend web development. That's okay. Creative coding with Processing's Python Mode need not be the ultimate goal of reading this book. Think of it as a starting point for exploring Python, other frameworks, creative applications for coding, and programming in general.

Where Can I Find Help?

Programming is rewarding, in part because it's challenging. If you find yourself struggling with something, do not stress; that's normal! With a little perseverance, you'll soon grasp whatever has you snagged.

Online Resources

If you're getting nowhere, reach out to online communities. You'll find a dedicated category for Processing.py in the official Processing forum at *https://discourse.processing.org/.* You'll often discover that somebody has already encountered and received a solution for the challenge you're facing; if not, go ahead and create a new topic. Incidentally, the author of this book is known to lurk about in this friendly and welcoming corner of the internet.

The official Python Mode reference is available at *https://py.processing .org/reference/.* Each entry includes a description and brief code example. It's handy to keep this web page open while you work in the Processing development environment.

Source Code and Solutions

You'll be typing a lot of code. This is a good thing because the best way to learn is by doing. At times, however, you might mistype something or be unable to figure why your code refuses to work. In such instances, it can be helpful to have access to a complete, working version of the file. You can access all of the code in this book, as well as solutions to the challenges in each chapter, at *https://github.com/tabreturn/processing.py-book/.* You can also find any updates to this book at *https://www.nostarch.com/ Learn-Python-Visually/.*

What's in This Book?

This book begins with the basics and builds up toward more advanced topics as you progress. Each chapter, therefore, requires a grasp of the concepts introduced in the chapters preceding it. You'll be working, step by step, through a series of practical tasks. You'll also find some theory, plenty of visuals, and challenges to consolidate what you've learned.

The following outline provides a brief overview of the contents in each chapter:

Chapter 1: Hello, World!   This chapter covers the installation and setup procedure for the book and introduces the basics of drawing with code. You'll also learn how computers manage color, how you can store and reuse values (using variables), and how to perform basic arithmetic operations using Python.

Chapter 2: Drawing More Complicated Shapes   Having covered some drawing essentials in the first chapter, you'll move on to drawing more organic shapes, as opposed to geometric ones. You'll learn to define shapes by using points (or vertices) and curves, which enable you to draw just about any shape with code.

Chapter 3: Introduction to Strings and Working with Text   In this chapter, you'll learn how to use Python's string features to manipulate text. You'll also learn how to use Processing functions to draw text to the display window, in different styles and colors, and in different fonts.

Chapter 4: Conditional Statements   This is where you really begin to think like a programmer. In this chapter, you'll introduce control flow to your programs. In other words, you'll learn how to write programs that can make decisions, executing different actions to respond to different situations.

Chapter 5: Iteration and Randomness   In this chapter, you'll learn how to write programs that can repeat an operation a specified number of times or until a certain requirement is met. Toward the end of the chapter, you'll experiment with randomness and creating tiled patterns.

Chapter 6: Motion and Transformation   This chapter focuses primarily on adding motion to your Processing programs and transforming the drawing space. You'll also learn how to save frames as images and how to get time values from your computer. You'll use these skills to create an animated screensaver and analog clock.

Chapter 7: Working with Lists and Reading Data   Python lists will unlock powerful ways to manage and manipulate values in collections. You'll explore techniques for data visualization. You'll also learn to read in list data from external files. For the final task, you'll render a chart by using a CSV file.

Chapter 8: Dictionaries and JSON   Dictionaries are similar to lists in that they store collections of items. With dictionaries, however, you access items by using a key (usually a word) instead of referring to the item position. Once again, you'll get to use your new dictionary skills for data visualization. You'll also learn to work with JSON data.

Chapter 9: Functions and Periodic Motion   You'll use functions to divide a program into named sections of reusable code. This will make your code more modular, and easier to read and modify. You'll also delve into some trigonometry for generating elliptical and wave-type motions.

Chapter 10: Object-Oriented Programming and PVector   You can use object-oriented programming to structure programs by modeling real-world objects. In this chapter, you'll employ an object-oriented approach to building an amoeba simulation. You'll also learn to program the amoebas' motion by using Processing's `PVector` class.
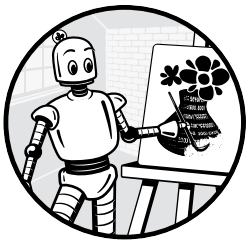
Chapter 11: Mouse and Keyboard Interaction   In this chapter, you'll add interactivity to your programs. Processing can handle input from various devices, but here you'll focus on mouse and keyboard input to build a paint app. In the process, you'll learn about event functions and how to control Processing's draw loop behavior.

Let's Go!

The speed at which you progress through these chapters is likely to be influenced by your prior experience in similar areas. If you've done any type of programming before, Python or otherwise, you'll encounter some familiar concepts. That said, it's not a race! Enjoy the ride, stop for breaks, and if you're feeling really inspired, feel free to head off-road.

1

HELLO, WORLD!

When learning a new programming language, it's a long-standing tradition that the first code you write is to display the message 'Hello, World!' In keeping with that tradition, you'll do that here too—but that's not all. This chapter introduces everything you need to understand the fundamentals of Processing, and you'll quickly move on from a simple 'Hello, World!' to drawing with code.

To get started, you'll set up Python Mode for Processing so you can create your own sketches. Along the way, you'll learn the basic rules of writing code in Processing, as well as how to deal with errors, use variables, and perform arithmetic operations. You'll also learn about how Processing handles color and how to measure angles using radians. By the end of this chapter, you'll be comfortable drawing colorful geometric shapes by using various Processing functions. Let's get started.

Processing Installation and Python Mode Setup

Before writing any code, you first need to set up Python Mode for Processing. Head over to the Processing downloads web page (*https://processing.org/download/*) and grab the version of Processing appropriate for your system (Windows, Linux, or macOS). As of January 2021, Processing 3.5.4 is the latest stable release.

Processing does not employ an installation program. Instead, you simply extract the file you have downloaded (usually a *.zip* archive) and run the application. The exact process varies slightly between operating systems:

- On Windows, unzip all of the contents by right-clicking the file and selecting **Extract All**, and then follow the instructions. Extract or move the folder to any location on your computer, including your *Program Files* folder or *Desktop*.
- On macOS, unzip the file by double-clicking it, and then move the extracted app to any location on your computer, including your *Applications* folder or *Desktop*.
- The Linux version of Processing is a *.tar* archive. Extract or move the folder to any location on your computer, including your home folder or desktop.

Once you're finished, open the newly extracted folder. Figure 1-1 shows an abridged listing of what you can expect to see in your file manager. Next, locate and run the executable file named *processing*. On macOS, you'll just have a single file named *processing*.
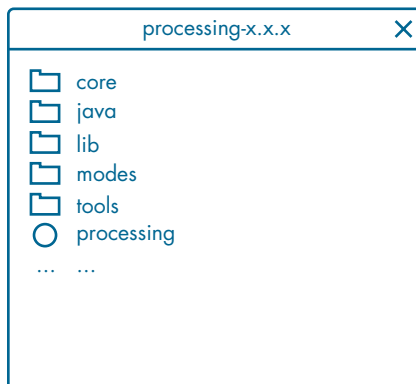
Figure 1-1: The contents of a freshly extracted processing *folder for Windows or Linux*

The application layout may vary slightly among systems and Processing versions, but the key elements are outlined in Figure 1-2. If you're a Mac user, you'll find the *menu bar* in its usual position at the top of your screen. Note that the upper right button in the Processing interface is labeled *Java*. This is because Processing comes bundled with Java mode as the default.
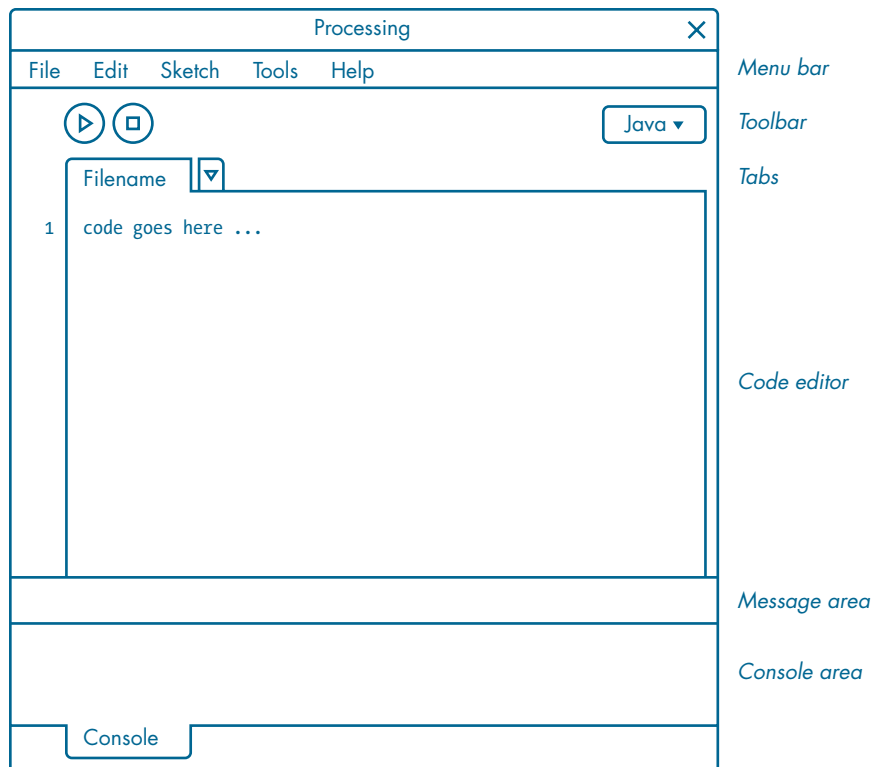
Figure 1-2: The Processing interface

Next, activate Python Mode. Click the **Java** button and select **Add Mode** from the drop-down menu; then, from the Contribution Manager window that appears, choose **Python Mode for Processing**. Finally, click **Install**. You can now change between Python and Java mode by using the drop-down menu. Switch to Python (Figure 1-3).
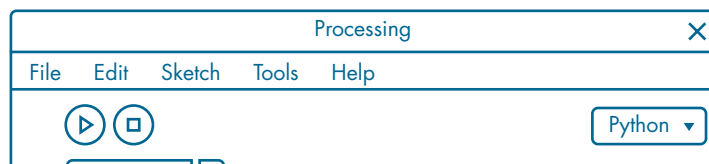
Figure 1-3: The button to the right indicates that Python Mode is activated.

You're now ready to write your first lines of code!

NOTE   *For a demonstration of what Processing can do, take a look at the Python examples (File ▶ Examples) included with Processing. To run any example, use the play (▶) button.*

Your First Sketch

Processing refers to programs as *sketches*. Given the visual and artistic nature of what you are likely to produce, it's a fitting term. Select **File ▸ New** to create a new sketch, or use the associated keyboard shortcut (listed alongside the menu entry).

Enter the following lines of code:

```
size(500, 500)
print('Hello, World!')
```

I'll go through the specifics of this code in a bit. For now, save the sketch by using **File ▸ Save As** and name it *hello_world*.

You will notice that Processing creates a new folder named *hello_world*; within it, there are two files: *hello_world.pyde* and *sketch.properties* (Figure 1-4). Depending on your system's configuration, you may or may not see the file extension (*.pyde*). To reopen any sketch, locate and open the *.pyde* file.

Figure 1-4: The contents of your hello_world *sketch folder*

You may want to add other assets to your sketch folders, such as images and fonts, but more on that later.

Next, click the play (▸) button to execute the code. Better yet, use the associated keyboard shortcut: CTRL-R for Windows and Linux, or ⌘-R for macOS. A gray 500 × 500 pixel *display window* should appear. In the console, which is the black area at the bottom of the editor, Processing should display `Hello, World!` (Figure 1-5).

Now let's get back to the code you entered in this file; it uses two Processing functions: `size()` and `print()`. *Functions* are named instructions, sort of like dog commands for computers. Some commands are simple, like "sit," but something like "fetch" may involve specifying what it is that Fido should retrieve.

Python functions consist of a function name followed by opening and closing *parentheses*, which is where you provide arguments. In the case of my dog-command analogy, "ball" could be an argument for "fetch." The `size()` function (Figure 1-6) takes two arguments: the first represents the width of your sketch, and the second is the height.

In this case, the display window is 500 pixels wide by 500 pixels high.
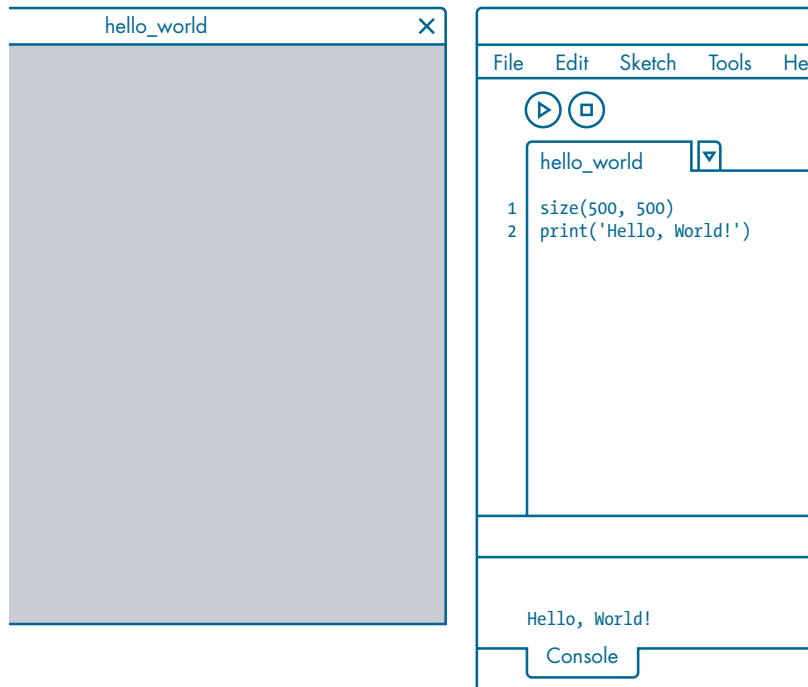
Figure 1-5: Display window (left) and an editor with 'Hello, World!' displayed in the console (right)

In the editor:
```
1  size(500, 500)
2  print('Hello, World!')
```

Console output: `Hello, World!`
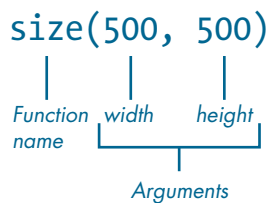
Figure 1-6: Anatomy of a size() function

The print() function writes to the console. This function takes a single argument: the phrase 'Hello, World!' Because this is text—or technically, *string* data—you need to wrap it in quotation marks. You can use single or double quotes, but be sure to use the same type for both opening and closing quotes.

Python classifies every value according to a *data type*, which determines how the value is handled and the operations you can perform on it. For example, you can perform arithmetic operations—like division or subtraction—on numeric data types but not on strings. In this chapter, you'll deal with three data types:

String   Text data, like 'Hello, World!'

Integer   Numbers without decimal points, such as 1, –27, or 422

Floating-point   Numbers that include a decimal point, such as 1.618

What separates Processing code from standard Python code are some of its functions; the size() function, for example, is Processing-specific. In other words, it won't work outside the Processing environment. The print() function, on the other hand, is a built-in element of the standard Python programming language. It works in Processing's Python Mode and any other Python program.

Throughout this book, I usually refer to Processing-exclusive features in the context of *Processing*, and standard Python features with the term *Python*. If this is confusing, think of them as one and the same. At this point, distinguishing Processing from Python isn't crucial; you will understand the differences in time.

HOW DO I KNOW WHAT ARGUMENTS TO PROVIDE?

The type, number, and order of arguments vary according to the function, and some functions don't require arguments. For a complete list of Processing functions and the arguments they require, as well as many standard Python elements, refer to the Processing.py reference at *https://py.processing.org/reference/*.

This book covers a lot of that online content, but the reference should be your go-to source for in-depth descriptions of each function.

Comments

If you want Python to ignore any part of your code, you can comment it out. This feature is useful for leaving notes, in plain English, to yourself or anybody else editing your code. Let's add a few comments to your *hello_world* file:

```
❶ # dimensions of the display window measured in pixels
size(500, 500)
print('Hello, World!')  # writes hello world to the console area
❷ '''
This is a multiline comment.
Any code between the opening and closing triple-quotes is ignored.
'''
print('How are you?')
```

Comments come in two types: single- and multiline. As shown here, use a # character for single-line comments ❶ and ''' (or """) for multiline comments ❷.

While working through the tasks in this book, add comments to remind yourself how the code works. Comments are also useful for debugging code. For instance, if you suspect that some lines are causing your program to fail, you can temporarily disable them by commenting them out.

Whitespace

Python, and by extension Processing's Python Mode, is whitespace-sensitive. You need to be careful where you insert space characters or tabs. As an example, add a few spaces to the beginning of the size() line; then run the sketch:

```
# dimensions of the display window measured in pixels
    size(500, 500)
print('Hello, World!')  # writes hello world to the console area
. . .
```

When you run the sketch, the message bar turns red, and Processing displays an error message (Figure 1-7). Python relies on indentation to distinguish blocks of code. The indented line breaks the program because Python did not encounter any code to define a new block for the size() function.
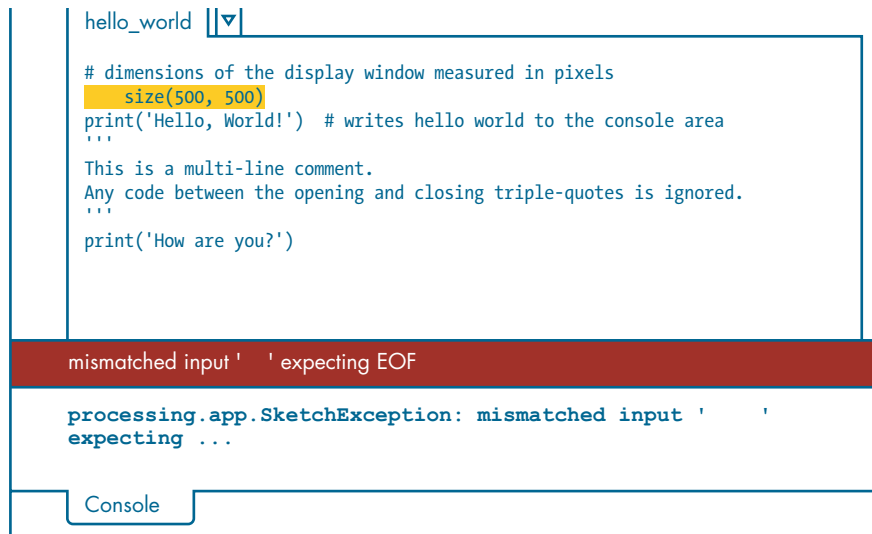
Figure 1-7: A whitespace error

Correct the code by removing the problematic space characters you've just added.

You'll come to understand more about when and where to use indentation as you progress through these chapters. For now, though, pay careful attention to any space and tab characters that affect the indentation of your code.

Errors

Whitespace issues are not the only error type you will encounter. On occasion, you're likely to miss the odd parenthesis, comma, or quotation mark, especially when starting out. Try removing the closing parenthesis from your `size()` function, like so:

```
# dimensions of the display window measured in pixels
size(500, 500
print('Hello, World!')  # writes hello world to the console area
. . .
```

Now run the code and observe the console output. Note the suggestion in the message bar (Figure 1-8). Pretty smart, huh?

```
hello_world  |▼|

# dimensions of the display window measured in pixels
size(500, 500
print('Hello, World!')  # writes hello world to the console area
'''
This is a multi-line comment.
Any code between the opening and closing triple-quotes is ignored.
'''
print('How are you?')



Maybe there's an unclosed paren or quote mark somewhere before this line?

processing.app.SketchException: Maybe there's an unclosed paren or...


Console
```

Figure 1-8: The alert in the message bar (red) suggests a possible cause for the error that Processing encountered.

This is an example of a *syntax error*, and it's not the first you'll encounter. Just as English sentences must begin with a capital letter and end with a full stop, Python functions must have an opening and a closing parenthesis. When you have multiple arguments, you need to separate them using commas. This set of rules is called *syntax*. If you don't conform to the rules, Python will be confounded and report an error.

Error messages are not always so clear or accurate, but they can provide a clue as to where to start searching for bugs. Copying and pasting the messages into a search engine can sometimes help you find a solution.

Color

You can describe colors in various ways in Processing. To keep things simple, I'll stick with *hexadecimal* values for the first example. If you're familiar with

graphics software like Adobe Photoshop, Adobe Illustrator, Inkscape, or GIMP, you may have seen hexadecimal values in the color picker for those programs.

Processing includes its own color selector (Figure 1-9), which you can access from the menu bar by selecting **Tools ▸ Color Selector**. You can use this color selector to mix and sample color values. The value that begins with a hash mark (#) is the hexadecimal; you use the **Copy** button to copy it, so that you can paste it into the code editor.
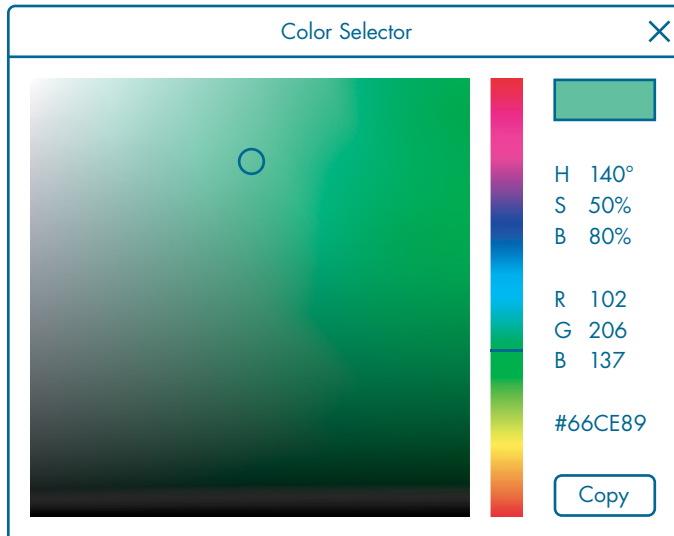
Figure 1-9: Processing color selector

Your screen displays color pixels by mixing three primary colors—much as you mix red, yellow, and blue paint in art class. However, your screen relies on red, green, and blue primaries instead. Furthermore, because light blends color in an *additive* manner, pixels that combine all three primaries at full intensity appear as white. Conversely, a complete absence of any color results in a black pixel. Other colors contain varied quantities of red, green, and blue. For example, a bright red mixture is created as follows:

100% | 0% | 0%

A hexadecimal color value is composed of six hexadecimal digits (0, 1, 2, . . . , 9, A, B, C, D, E, F) and can be split into three pairs. Each pair corresponds to a primary color. Here's the value for bright red:

#FF0000

The `FF` represents a red quantity; the middle `00` is for green; the rightmost `00` is for blue. For reasons I won't get into here, `FF` is the equivalent of 100 percent. Also, remember that you are mixing light, so #FFFFFF is white, and #000000 is black. Here are some other examples:

100 percent blue   #0000FF

Dark green   #006600

Dark gray   #505050

Use the selector to experiment further and observe how the hexadecimal values change as you select different colors.

The `fill()` function sets the color used to fill shapes. It accepts up to four arguments, depending on the color system you are using. For hexadecimal color, use a single argument: the six-digit value prefixed with a #, wrapped in quotes.

Add the following line to the bottom of your *hello_world* sketch:

```
. . .
fill('#FF0000')
```

You have now set the fill color to red. To see this in effect, let's draw a rectangle. The `rect()` function is used for drawing rectangles, and it takes four arguments:

```
rect(x_coordinate, y_coordinate, width, height)
```

The first two arguments specify the location of the rectangle's upper left corner (Figure 1-10). Processing's x-coordinates begin from the left edge of the display window; the y-coordinates begin from the top edge.
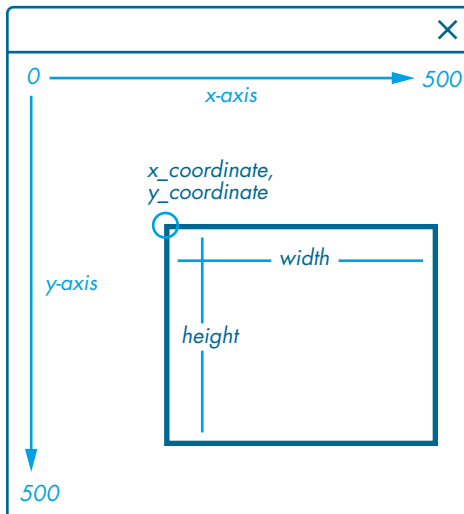
Figure 1-10: Processing's coordinate system

The x-y coordinate for the upper left corner of the display window is (0, 0), and the lower right is (500, 500). So to move the rectangle down, increase the y-coordinate value. Add a new rectangle line to your *hello_world* sketch:

```
. . .
fill('#FF0000')
rect(100, 150, 200, 300)
```

Run the sketch to confirm that the output matches Figure 1-11. Experiment with the rect() arguments to affect the rectangle's size and position.
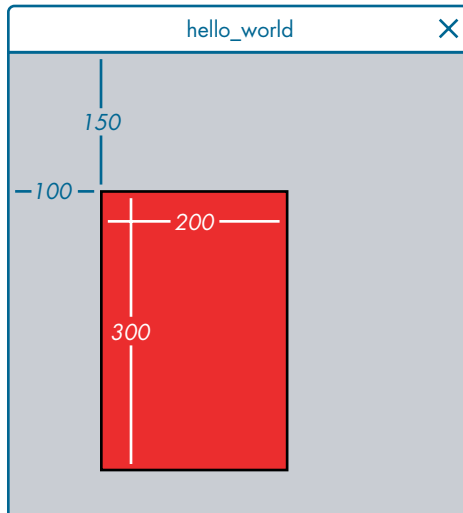
Figure 1-11: rect(100, 150, 200, 300)

You now should be familiar with Processing's coordinate system. The rect() is one of many drawing functions; you'll be introduced to a few more shortly.

In this section, you've also learned to define color using hexadecimal values that describe different quantities of red, green, and blue light. And you now can use a color picker, like the one included with Processing, to mix and sample any values you require. You'll see other systems for defining color in Processing, but for most of this book, you'll use hexadecimal.

Fills and Strokes

When you write a fill() line, every shape thereafter is filled in with the color you specify. That color won't change until Processing encounters the next fill() line. In this way, Processing is like painting: you grab a brush and dip it in paint, and then everything you paint is influenced by the brush and color you last selected. When you want to paint in a different style or color, you change out your brush or dip it in a different pot. If you want to disable the fill altogether, use noFill().

Add the following code to the end of your *hello_world* file to draw a smaller red rectangle, an orange square, and a square with no fill:

```
. . .

# small red rectangle
rect(10, 15, 20, 30)

# orange square
```

```
   fill('#FF9900')
❶ rect(50, 100, 150, 150)

   # fill-less square
   noFill()
❷ square(250, 100, 150)
```

For a square, you have two options: use a rect() ❶ with matching width and height (third and fourth) arguments. Or, use the square() ❷ function, which takes three arguments: x, y, and extent.

Processing interprets lines of code from top to bottom. As a result, shapes at the bottom of your code appear at the top of the visual "stack." So the previous code produces the shapes in Figure 1-12.
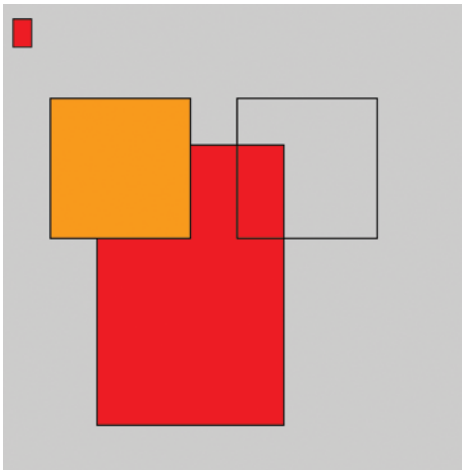
Figure 1-12: The no-fill square—the last line of your code—is the topmost shape.

Stroke is another term for *outline*, and you're likely to use the following three stroke functions: stroke() to change the color, strokeWeight() to change the width, and noStroke() to disable the stroke altogether. Like fill() and noFill(), the stroke functions affect everything below them.

For a white stroke, 3 pixels in width, insert the following lines above the shape code:

```
. . .
stroke('#FFFFFF')
strokeWeight(3)

fill('#FF0000')
. . .
```

The `stroke()` line affects every shape that follows it. Figure 1-13 shows how all of the shapes now have white strokes.
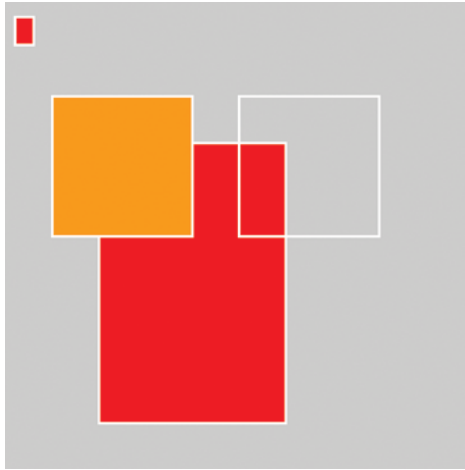
Figure 1-13: Adding white strokes

For thicker strokes, you may want to specify whether the corners and tips are rounded or sharp. For more information, consult the relevant Processing.py reference entries for `strokeCap()` and `strokeJoin()`.

Background Color

To change the background color, use the `background()` function. Add a background line to the end of your sketch:

```
. . .
square(250, 100, 150)
background('#004477')
```

Run the sketch and note how everything has disappeared; the entire display window is now a flat shade of blue. This is because `background('#004477')` draws over everything before it, which will be useful when you start working with animation. For now, move that line to the top of your code so you can see the shapes again (Figure 1-14):

```
# dimension of the display window in units of pixels
size(500, 500)
background('#004477')
. . .
```

Note that the background function can also accept an image as an argument (I'll introduce images in Chapter 2).
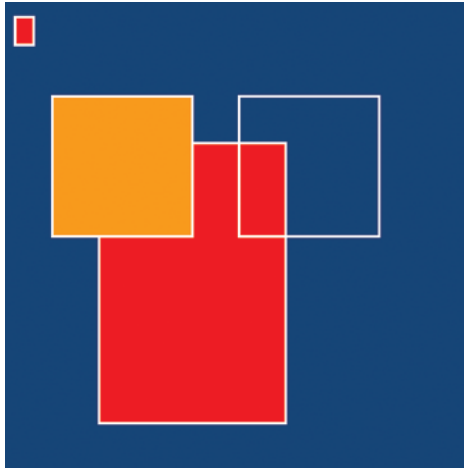
Figure 1-14: Adding a background color

Color Modes

I'll use hexadecimal color values for the rest of the chapter, but here's a quick introduction to other color modes, because at some point, you may need to express colors in something other than hexadecimal. For example, say you want to write code that darkens a bright red fill. First, you will recall that this is a shade of bright red:

```
fill('#FF0000')
```

You can also represent this color as the following in RGB:

```
fill(255, 0, 0)
```

In this arrangement, each red/green/blue value is comma-separated. As you might have already deduced, 255 is equivalent to FF (which itself is equivalent to 100 percent). To make the red half as bright, you can subtract 127 from 255. However, trying to subtract 127 from FF is tricky because you're dealing with a mix of hexadecimal and decimal numbers. In this instance, it's easier to stick with decimal values (255 – 127 = 128).

To use the fill(255, 0, 0) notation, Processing must have its colorMode() set to RGB. You don't need to specify this, though, as it's the default mode. Here's how it works: if Processing detects a single argument in quotes (like '#FF0000'), it interprets it as hexadecimal, but if you provide three arguments, it automatically knows that you are using the system of 0 to 255.

However, you can use another mode: HSB. Once set to HSB mode, the three fill() arguments represent *hue*, *saturation*, and *brightness*, respectively. To better understand how those variables affect color, let's take a look at the color picker of GIMP, an open source image editor (Figure 1-15).

Rotating the large triangle adjusts the H value between 0 and 360 degrees; the H (hue) field corresponds to the white line at the triangle's lower right tip. You move the small white circle (inside the triangle) to adjust the S (saturation) and V (value) fields. *Value* and *brightness* are interchangeable terms in this context, so the V corresponds to the B in HSB.
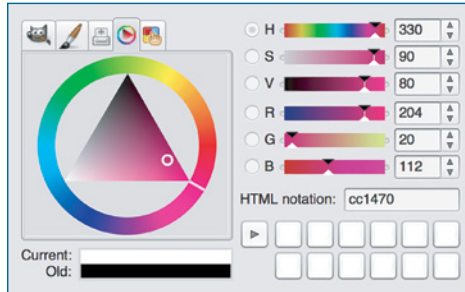
Figure 1-15: Hue: 330 degrees; saturation: 90 percent; value/brightness: 80 percent

If you have GIMP installed, or software with a similar color picker, I encourage you to experiment with it. To mimic the GIMP scheme in Processing, set the color mode accordingly:

```
colorMode(HSB, 360, 100, 100)
```

The HSB represents the mode, 360 represents the range of degrees for hue, and the two 100 arguments represent a range of 0 to 100 percent for saturation and brightness. You would now write a red fill as follows:

```
fill(0, 100, 100)
```

This is because bright red lies at 0 degrees of rotation on the hue ring (which begins at "East" in the GIMP mixer) and has a saturation and brightness of 100 percent (Figure 1-16).
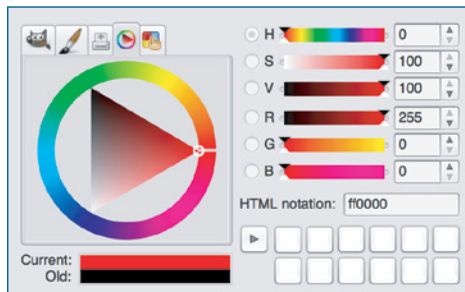
Figure 1-16: Hue: 0 degrees; saturation: 100 percent; value/brightness: 100 percent

In HSB mode, shifting along the color spectrum—from red to orange to yellow to green, and so on—is a simple matter of adding to or subtracting from the H value. Attempting the same in RGB mode is not so easy, as you need to adjust the proportions of each primary color.

The chapters to come will cover even more on color. Consult the relevant Processing.py reference entries for `colorMode()` and `fill()` if you need more detail.

2D Primitives

Let's move on to drawing basic shapes. Begin a new sketch (**File ▸ New**) and save it as *primitives_2d* (**File ▸ Save As**). Add this code to set things up before proceeding:

```
size(600, 300)
background('#004477')
noFill()
stroke('#FFFFFF')
strokeWeight(3)
```

Now when you run the sketch, thanks to `background('#004477')`, an empty blue display window appears. Any shapes that you draw will have no fill and a white stroke of 3 pixels.

Next, draw three points by using the `point()` function (Figure 1-17):

```
point(100, 25)
point(200, 25)
point(150, 75)
```

The `point()` function accepts two arguments, which represent the x- and y-coordinates. The active `strokeWeight()` determines the size of the points.

Figure 1-17: Three points drawn with the `point()` function

What follows are descriptions for several drawing functions, along with code to add to your working sketch. Experiment with the arguments to see how the finished version (Figure 1-18) responds.
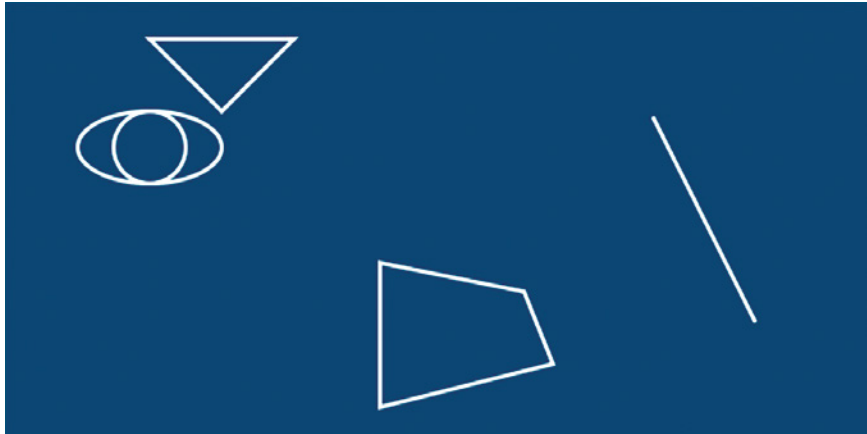
Figure 1-18: An assortment of 2D primitives

triangle()

The `triangle()` function draws a triangle. The six arguments represent three x-y coordinate pairs. I have grouped each x-y pair by removing the space character after every other argument so it's easier to read:

```
triangle(100,25, 200,25, 150,75)
```

Python is not sensitive to whitespace between arguments, so if you find it helpful to format your code in a similar fashion, feel free to do so.

ellipse()

The `ellipse()` function draws an ellipse. The first pair of arguments represents an x-y coordinate that marks the center of the ellipse; the second pair of arguments represents the ellipse's width and height:

```
ellipse(100,100, 100, 50)
```

For a circle, you can use the `ellipse()` function with matching width and height (third and fourth) arguments. Alternatively, you can use the `circle()` function, which takes three arguments: x, y, and diameter.

```
circle(100,100, 50)
```

The `circle()` and `square()` functions are relatively new to Processing, so you may find that many of the examples (**File ▸ Examples**) and online code rely on only `ellipse()` and `rect()`.

quad()

The quad() function draws a *quadrilateral* (a four-sided polygon). Essentially,
it's like a triangle function with an extra point, and its eight arguments represent four x-y coordinate pairs:

```
quad(260,180, 360,200, 380,250, 260,280)
```

line()

The line() function draws a straight line between two points. The first pair of
arguments represents the starting x-y coordinates, and the second pair, the
ending x-y coordinates:

```
line(450,80, 520,220)
```

As with points and shapes, the width of a line is affected by any preceding strokeWeight() function.

The 2D primitive functions provide an easy way to draw shapes in the
display window. There's one more shape function to review, arc(), but it's a
bit more involved than the other shapes. Variables and arithmetic operators
will prove useful for drawing arcs, so I'll cover those first. Before moving
on, though, here's a quick challenge to practice what you've learned so far.

Challenge #1: Rainbow Task

Begin a new sketch (**File ▸ New**) and save it as *rainbow* (**File ▸ Save As**). Add
this code to get started:

```
size(600, 300)
background('#004477')
noStroke()
```

Using what you've learned so far, complete the rainbow in Figure 1-19.

Clue: think about how you can overlap shapes to mask others. If you
need help, you can access the solution at *https://github.com/tabreturn/
processing.py-book/tree/master/chapter-01-hello,world!/rainbow/*.

Figure 1-19: Recreate this rainbow.

Variables

Variables are placeholders for information—much like when you use letters in algebra to represent a value. In fact, Python variables look and behave quite similarly.

Begin a new sketch and save it as *variables*. To keep things simple, you'll print values to the console area. Add the following code to set up the sketch and print its width and height (in pixels):

```
size(600, 400)
background('#004477')
noStroke()

print(width)
print(height)
```

If you run the sketch, the display window's width and height should be printed to the console, as shown in Figure 1-20.

But notice that you never explicitly defined width or height. Processing automatically assigned the width and height of the display window to these two variables. From this, you can establish that width and height are variables for which Processing maintains the values. Predefined variables like these are called *system variables.*

However, you are not limited to system variables; you can also define your own. When declaring new variables, assign them a value by using an equal sign (=), which is called the *assignment operator.* Try this out with a new variable named x:

```
. . .
x = 10
print(x)  # displays 10 in the console
```
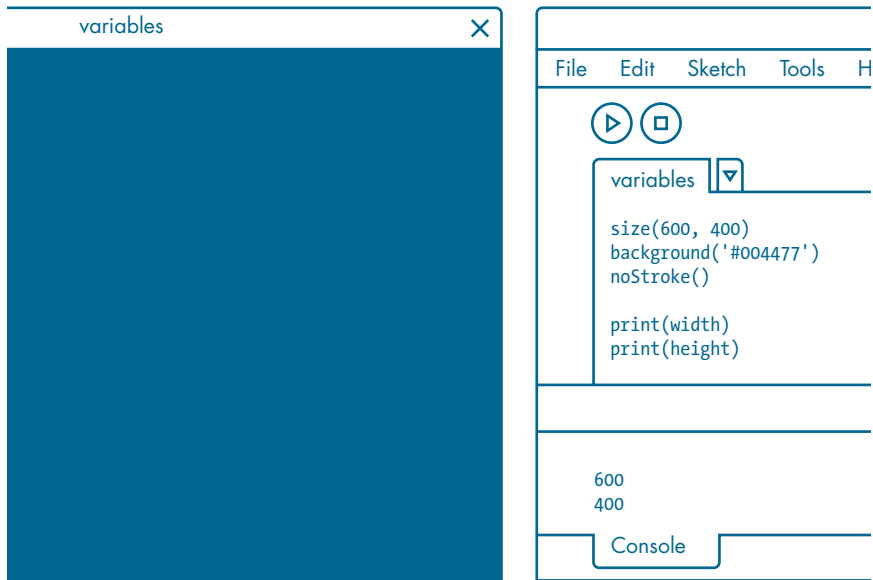
Figure 1-20: Printing variables to the console

The variable x is equal to 10, so the `print()` function displays a 10 in the console.

You can name your variables whatever you like, provided that the name contains only alphanumeric and underscore characters, doesn't begin with a number, and doesn't clash with any reserved keywords or variables (like `width`). For example, the following shows several possible variable names (the comments indicate which are correct):

```
playerlives = 3    # correct
playerLives = 3    # correct
player_lives = 3   # correct
player lives = 3   # incorrect (contains a space)
player-lives = 3   # incorrect (contains a hyphen)
player2lives = 3   # correct
2playerlives = 3   # incorrect (begins with a number)
```

Whether you should name a multiword variable using *camelCase*, underscores, or another convention is a matter of style (and vociferous debate), but it's good to decide on a naming convention and stick to it, as you'll make extensive use of variables in Processing.

NOTE *The camelCase convention is used to form one word from multiple words; each new word except the first starts with a capital letter. In this way, you can tell words apart without using spaces. You'll use the camelCase convention for naming user-defined functions (Chapter 9) and the UpperCamelCase convention for objects (Chapter 10).*

Now, add three more variables to your script for use as arguments in a rect() function:

```
. . .
y = 30
w = 100
h = w
rect(x, y, w, h)
```

The y variable indicates the y-coordinate; w indicates the width; and h indicates the height value for the rect() function. Notice that the h value is equal to the w value (of 100). You've already defined x as equal to 10. Run the sketch to confirm that it displays a white square positioned near the upper left corner of the display window (Figure 1-21).

Figure 1-21: A square using variables as coordinates

Experiment further on your own with shapes and variables. In the next section, you'll learn to use variables to perform mathematical calculations.

Arithmetic Operators

Arithmetic operators perform arithmetic operations on *operands*; this is far simpler than it probably sounds. For instance, in the expression 1 + 3, the plus sign is the operator, and the numbers 1 and 3 are the operands. To better understand how this all works in Python, let's go through some examples.

Basic Operations

Add the following line to the end of your *variables* sketch to calculate what variable x plus 2 is equal to:

```
. . .
print(x + 2)
```

Run the sketch and check the bottom line of the console output. I'm guessing the code did exactly what you expected. Earlier in the chapter, you assigned a value of 10 to variable x, and 10 + 2 is equal to 12, which is what you will see in your console.

You can also subtract (see the comments for the result):

```
. . .
print(x + 2)        # displays 12
print(x - 2)        # displays 8
```

Use the * operator for multiplication:

```
. . .
print(x * 2)        # displays 20
```

Now try this line, but before running it, see if you can predict the result:

```
. . .
print(1 + 2 * 3)     # displays ???
```

The console displays a 7 and not a 9 because multiplication occurs before addition. Certain operators take precedence over others. Remember PEMDAS? It's an acronym to help you recall the *order of operations*, which is parentheses first, then exponents, then multiplication/division, then addition/subtraction. (Some people may be more familiar with the acronyms BEDMAS or BODMAS, which use the terms *brackets* for *parentheses,* and *of* or *order* to indicate exponents.)

If you want the addition to happen first, use parentheses:

```
print (1 + 2 * 3)   # displays 7
print((1 + 2) * 3)  # displays 9
```

For division, use a forward slash (/):

```
print(4 / 2)        # displays 2
```

Be aware that dividing two integers always produces an integer result (integers are whole numbers, as opposed to those with a decimal point). For example:

```
print(3 / 2)        # displays 1
```

Processing discards any decimal digits, effectively rounding down the result. Note, however, that this is Python 2 behavior. At the time of this writing, Processing's Python Mode uses Python 2.7. If you ever find yourself writing Python 3 code, the result will be 1.5.

For floating-point division in Python 2, define at least one of your operands by using a decimal point:

```
print(3 / 2.0)      # displays 1.5
```

This line displays a 1.5 in Python versions 2 and 3. This book avoids any Python code that isn't compatible with Python 3. Rest assured, you can apply your newly acquired coding skills in Python 2 and 3 development. Should Processing switch to Python 3, your code will still run fine.

Of course, division-by-zero operations will result in errors (Figure 1-22).

```
print(x * 2)        # displays 20
print(1 + 2 * 3)    # displays 7
print((1 + 2) * 3)  # displays 9
print(4 / 2)        # displays 2
print(3 / 2)        # displays 1
print(3 / 2.0)      # displays 1.5
print(3 / 0)        # error
```

ZeroDivisionError: integer division or modulo by zero

```
7
9
2
1
1.5
processing.app.SketchException: ZeroDivisionError: integer division ...
```

Console

Figure 1-22: Division-by-zero error

Processing uses other arithmetic operators (for floor division and exponents) that aren't necessary to review here. The modulo operator, however, warrants a brief introduction.

Modulo Operator

The *modulo operator* calculates the remainder of a division operation and is expressed as a percentage sign (%). Take 5 divided by 2 as an example. You could say the answer is 2.5, or you could say the answer is 2 remainder 1, because 2 "goes into" 5 twice with 1 left over.

The modulo operator performs the latter operation and provides the remainder. Here's some code contrasting division and modulus (as before, the comments show the output):

```
print(5.0 / 2)      # displays 2.5
print(5.0 % 2)      # displays 1.0
```

It may not be evident why this operator is useful. However, many important algorithms, such as those used in cryptography, use modular arithmetic. For now, consider that modulo operations resulting in a 0 indicate that numbers divide exactly. Among other uses, this is handy for establishing whether a number is odd or even:

```
print(7 % 2)        # displays 1, therefore 7 is odd
print(6 % 2)        # displays 0, therefore 6 is even
```

You'll use the modulo operator in the chapters to come.

Arcs

Now that I've covered variables and some basic math, I can introduce the arc() function, which is used to draw elliptical arcs. Let's look at a few examples to see how this one works. Create a new sketch and save it as *disk_space_analyzer*. Add the following setup code that will define some visual parameters to get started:

```
size(600, 600)
background('#004477')
stroke('#FFFFFF')
strokeWeight(3)
noFill()
```

The arc() function takes the following arguments, expanded across multiple lines here for easier comprehension (recall that Python is not sensitive to whitespace between function arguments):

```
arc(
  x_coordinate, y_coordinate,
  width, height,
  start_angle, end_angle
)
```

Add an arc to your sketch by using a *start_angle* of 0 and *end_angle* of 2:

```
. . .
arc(
  width/2, height/2,
  200, 200,
  0, 2
)
```

The green overlay in Figure 1-23 helps illustrate how the arguments work. Processing draws the arc along the perimeter of an invisible ellipse centered in the display window. The center of this ellipse has an x-y coordinate of width/2, height/2; it's 200 pixels wide and 200 pixels high. An angle of 0 is positioned at East, which opens clockwise to an angle of 2, which looks more like around 115 degrees of rotation.

The reason for this large angle is that Processing uses *radians* and not degrees to measure angles; 1 radian is equal to roughly 57.3 degrees. Why use radians? Radians, a standard unit of angular measure used in many areas of mathematics, provide more natural and elegant formulas for circular motion. Think about this: why are there 360 degrees in a full circle? Why not 300 or 100, or even a million degrees? While I'm on the topic, why are there 60 minutes in an hour? Or 24 hours in a day? Much of this has to do with ancient counting systems.
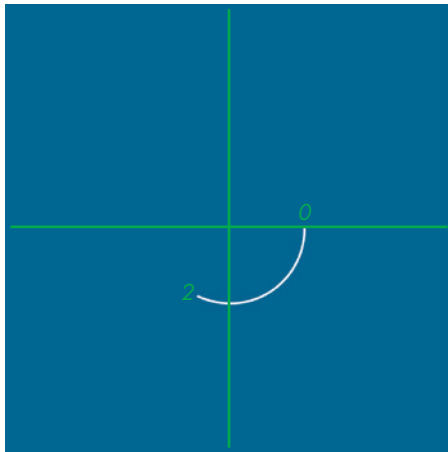
*Figure 1-23: An arc with a start angle value
of 0 and end angle of 2*

Rather than dividing a circle into an arbitrary number of slices (like 360), the radian system is based on a proportional measurement involving a circle's *radius*. Figure 1-24 illustrates how radians are defined. Beginning with the left graphic, take the radius of any circle; create an arc of that same length; then measure the angle formed between the tips of the arc and the center of the circle to derive one radian.
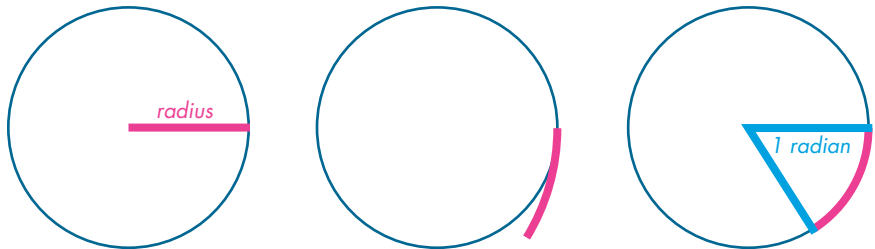
Figure 1-24: Defining a radian

If 1 radian is equal to approximately 57.3 degrees, 2 radians equal 114.6 degrees. This makes 180 degrees equivalent to roughly 3.142 radians (Figure 1-25). Do you recognize that number? Yep, it's pi!
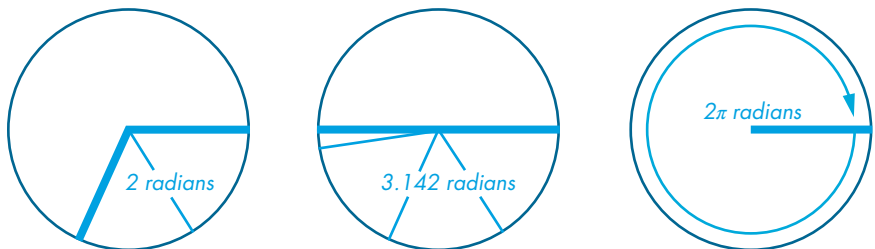
Figure 1-25: Measuring the number of radians in a half- and full circle

Processing provides the degrees() and radians() functions for convert-ing between the systems, but you should be fine working in radians if you can remember a few key measurements. For starters, 0 degrees is equal to 0 radians, and 180 degrees is equal to π radians. Therefore, 360 degrees is equal to 2π radians. In Processing, you can use the system variable PI instead of writing a lengthy decimal.

Add the following code to draw a half-circle and a full circle using arc() functions:

```
. . .
arc(width/2, height/2, 300, 300, 0, PI)     # half-circle
arc(width/2, height/2, 400, 400, 0, PI*2)   # full-circle
```

Run the sketch. The first new arc begins at 0 and ends at PI, resulting in a half-circle; the second outermost and largest arc has an end angle of PI*2, and therefore, it appears as a complete circle.

If you want to close an arc, so as to form a "slice," add an additional PIE argument. Add the following line to test this out:

```
. . .
arc(width/2, height/2, 350, 350, 3.4, (PI*2)-(PI/2), PIE)
```

The arc spans from 3.4 radians (around 10 o'clock) to ~4.7 radians (12 o'clock). Figure 1-26 depicts the final result. You can identify the most recent arc by its slice shape.

Figure 1-26: Here are four arcs, and one of them is a complete circle. The slice-like arc (upper left) uses the PIE argument.

Now for a final challenge before moving on to Chapter 2. A *disk usage analyzer* presents a graphical representation of a disk drive's contents. The Linux GNOME Disk Usage Analyzer (also known as *Baobab*) is one example of such software, and its charts make good use of arcs.

Recreate the ring chart graphic in Figure 1-27 by using what you have learned thus far. Begin by commenting out your existing arc lines, and then continue to work in the same sketch file. (The text and number labels have been added to assist you with calculations; do not add them to your recreation.)

Figure 1-27: Disk usage analyzer chart

If you need help, remember you can access all of the solutions to the challenges at *https://github.com/tabreturn/processing.py-book/*.

Summary

You now have Python Mode for Processing up and running. You also know how to set up a new sketch, set the size of your display window, and apply a background color. You've learned to display messages like 'Hello, World!' in the console and draw shapes using 2D primitive functions. You've also learned about color and how to define the color of your strokes and fills in hexadecimal, or using RGB and HSB color modes. In addition, you should understand how to use radians to measure angles and work with the `arc()` function.

While getting started with Processing, you've also learned a few Python programming fundamentals, like how to manage whitespace, add code comments, and use arithmetic operators to perform mathematical operations. You've also seen how to use Python variables, which are placeholders for data. Processing includes system variables, like `width` and `height`, but you can store values in your own variables, provided that the variable names adhere to Python's naming rules.

In Chapter 2, you'll learn how to draw more organic, as opposed to geometric, shapes. You'll also gain insight into the inner workings of vector graphics software like Adobe Illustrator and Inkscape.

2

DRAWING MORE COMPLICATED SHAPES
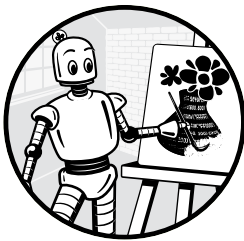
In Chapter 1, you learned about 2D primitives, including arcs, ellipses, lines, points, quads, rectangles, and triangles. However, some shapes, like hearts, stars, octagons, and Pikachu silhouettes, don't fit into any such category and require more than shape functions to create.

In this chapter, you'll learn how to draw more complicated shapes with points and curves, as well as vertex functions for laying points. Using these techniques, you'll draw shapes that blend straight and curved lines, and you'll create *negative* shapes by subtracting one shape from another.

You'll also learn how to work with two types of curves: *Catmull-Rom splines* and *Bézier curves*. Although both involve complicated math, Processing's curve functions handle the underlying calculus, allowing you to create curves with just the coordinates of a few control points.

Displaying a Grid

The best way to understand how curves work in Processing is to draw a few and then manipulate them. It's easier to plot points and curves by using a grid background for reference, so you'll add one by using a ready-made graphic. Create a new sketch and save it as *curves*, and then follow these instructions to download the grid graphic:

1. Open your web browser and go to *https://github.com/tabreturn/processing .py-book/*.
2. Navigate to *chapter-02-drawing_more_complicated_shapes*.
3. Download the *grid.png* file.

Additional sketch assets (images, fonts, and other media) belong in a subfolder named *data*, so create a new *data* subfolder within your *curves* sketch folder and place the *grid.png* file within it (Figure 2-1).
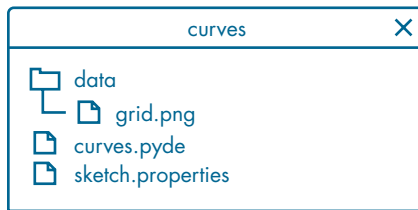
Figure 2-1: Place the grid graphic within your data *subfolder.*

This grid graphic will lie beneath everything you draw (Figure 2-2), assisting you in gauging x-y coordinates. Set up your sketch by using the following code:

```
size(500, 500)
❶ grid = loadImage('grid.png')
❷ image(grid, 0, 0)
  noFill()
  strokeWeight(3)
```

The `loadImage()` function loads the graphic file and assigns it to a variable named `grid` ❶. The `image()` function ❷ draws the image to the display window. The three arguments (`grid`, `0`, `0`) represent the loaded image file, x-coordinate, and y-coordinate, respectively.

The image is drawn at its original dimensions unless it's resized using an additional fourth (width) and fifth (height) `image()` function argument.
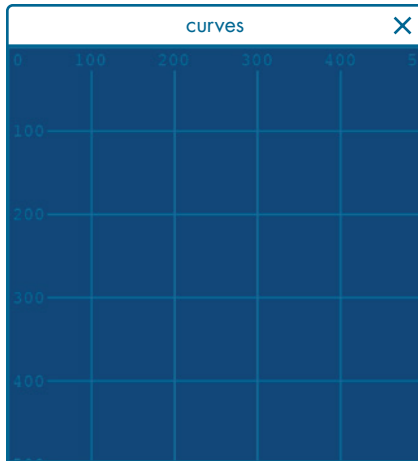
Figure 2-2: Displaying the grid image

Drawing Curves Using Catmull-Rom Splines

To draw a curved line in Processing, you can use the curve() function. This function accepts eight arguments, which represent four pairs of x-y coordinates; these are the starting control point, start of the curve, end of the curve, and ending control point.

Let's begin with a standard line and then adapt it into a curve. This way, you can visualize how the curve() function operates by comparing it with the simpler and more familiar line() function. Add a diagonal line to your *curves* sketch (Figure 2-3):

```
. . .
stroke('#0099FF')  # pale blue
line(100,100, 400,400)
```
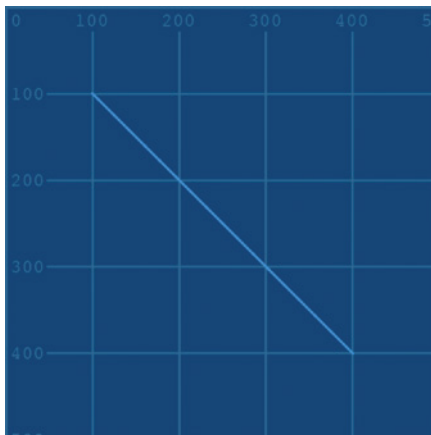

Figure 2-3: A straight line to adapt into a curve

Processing draws a line between the specified pairs of x-y coordinates: (100, 100) and (400, 400). Note that the line's coordinates correspond to the grid beneath.

CATMULL–ROM SPLINES

Processing's curve() function is an implementation of Catmull-Rom splines. Named after Edwin Catmull and Raphael Rom, a *Catmull-Rom spline* is a curve whose position and curvature depend on four points. The term comes from devices called *splines*, which are the long, thin, flexible strips of wood, plastic, or metal that draftsmen would use to draw smooth curves before they had computers.

Curving Lines with curve()

To use the curve() function to draw the same line, comment out the line() function in the *curves* sketch and replace it with a curve() function:

```
. . .
stroke('#0099FF')  # pale blue
#line(100,100, 400,400)
curve(0,0, 100,100, 400,400, 500,500)
```

When you run the sketch, the visual result should be exactly the same, as shown previously in Figure 2-3. The four middle values within the curve() function's parentheses match those of the line() function, and they also indicate the starting and ending x-y coordinates of the curve.

But the curve() function takes four additional outer arguments (in this example, 0,0 and 500,500), which represent two pairs of *control-point* coordinates. The positions of these control points determine the direction and amount of curvature you apply to the line. Before exploring this in detail, add the following new lines to the end of your code to draw a yellow line of the same length, at the same position, but with some curvature:

```
. . .
stroke('#FFFF00')  # yellow
curve(0,250, 100,100, 400,400, 500,250)
```

In this instance, the four middle arguments remain the same, but the control-point coordinates have been changed to 0,250 and 500,250. The result is a yellow curve with a slight S-bend (Figure 2-4). By comparing the blue and yellow lines, you can visualize how changing the control points has manipulated the curve.
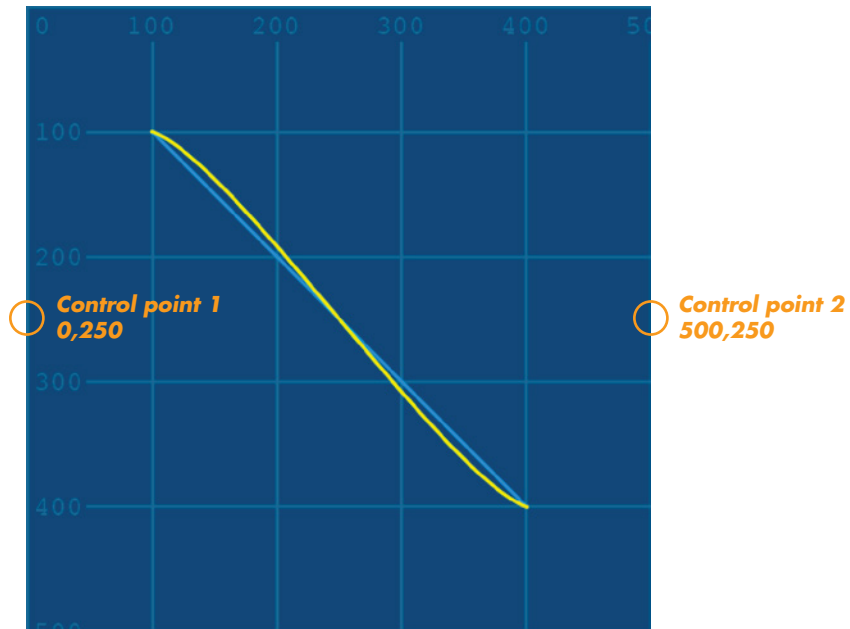
Figure 2-4: The yellow curve's control points, circled in orange, would otherwise be invisible.

To understand how the control points influence the curve, imagine that each end of the yellow curve extends to its neighboring control point. The closer you bring the control point to the center of the display window, the harder you are "flexing" this curve. Conversely, with control points 1 and 2 positioned at the upper left and lower left corners of the display window, respectively, the four points lie in a row, and the curve does not have to flex, resulting in a straight line.

To see how the control points work, add the following orange curves to serve as visual aids:

```
. . .
stroke('#FF9900')  # orange
# control point 1:
❶ curve(0,250, 0,250, 100,100, 400,400)
# control point 2:
❷ curve(100,100, 400,400, 500,250, 500,250)
```

The first curve() function ❶ draws an orange curve from control point 1 to the starting point of the yellow curve; the second curve() function ❷ draws another orange curve from the end point of the yellow curve to control point 2. The result (Figure 2-5) is a three-part curve (orange-yellow-orange) that shows how the control points determine the curvature of the yellow part.
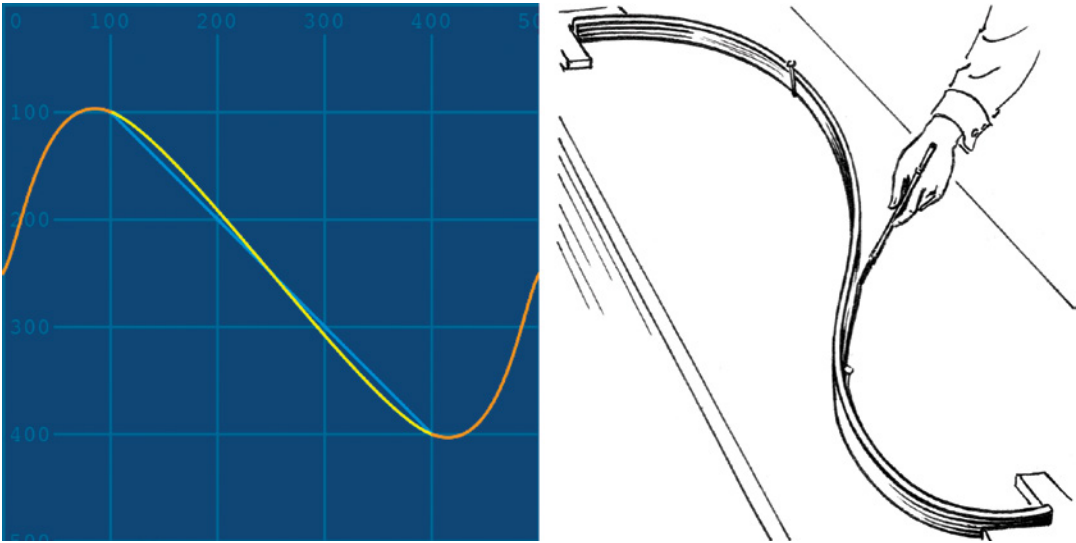
Figure 2-5: Your Processing curve (left) and a traditional spline (right). (Illustration: Pearson Scott Foresman, licensed under public domain.)

As you can see, the orange curves extend the yellow curve and illustrate what the yellow curve would look like as a physically complete spline. To the right in Figure 2-5, you can see the flexible strip for drawing such a curve without the aid of a computer. As mentioned earlier, it's this strip from which the spline takes its name. The two nails correspond to the starting and ending points of the curve() function, and the L-pieces at each end represent the control points.

Changing Curves with curveTightness()

The curveTightness() function determines how rigidly the curve conforms to the points that control it, as if you were replacing the draftsman's spline with a strip of less or more pliable material, or feeding a shorter or longer length of spline into the same area. The function accepts values ranging from –5.0 to 5.0, with 0 being the default.

To experiment, add a curveTightness() line above the yellow stroke:

```
. . .
curveTightness(0)   # try values between -0.5 and 0.5
stroke('#FFFF00')   # yellow
```

Enter different values to affect the curves below it. Figure 2-6 shows curves with different curveTightness() values.
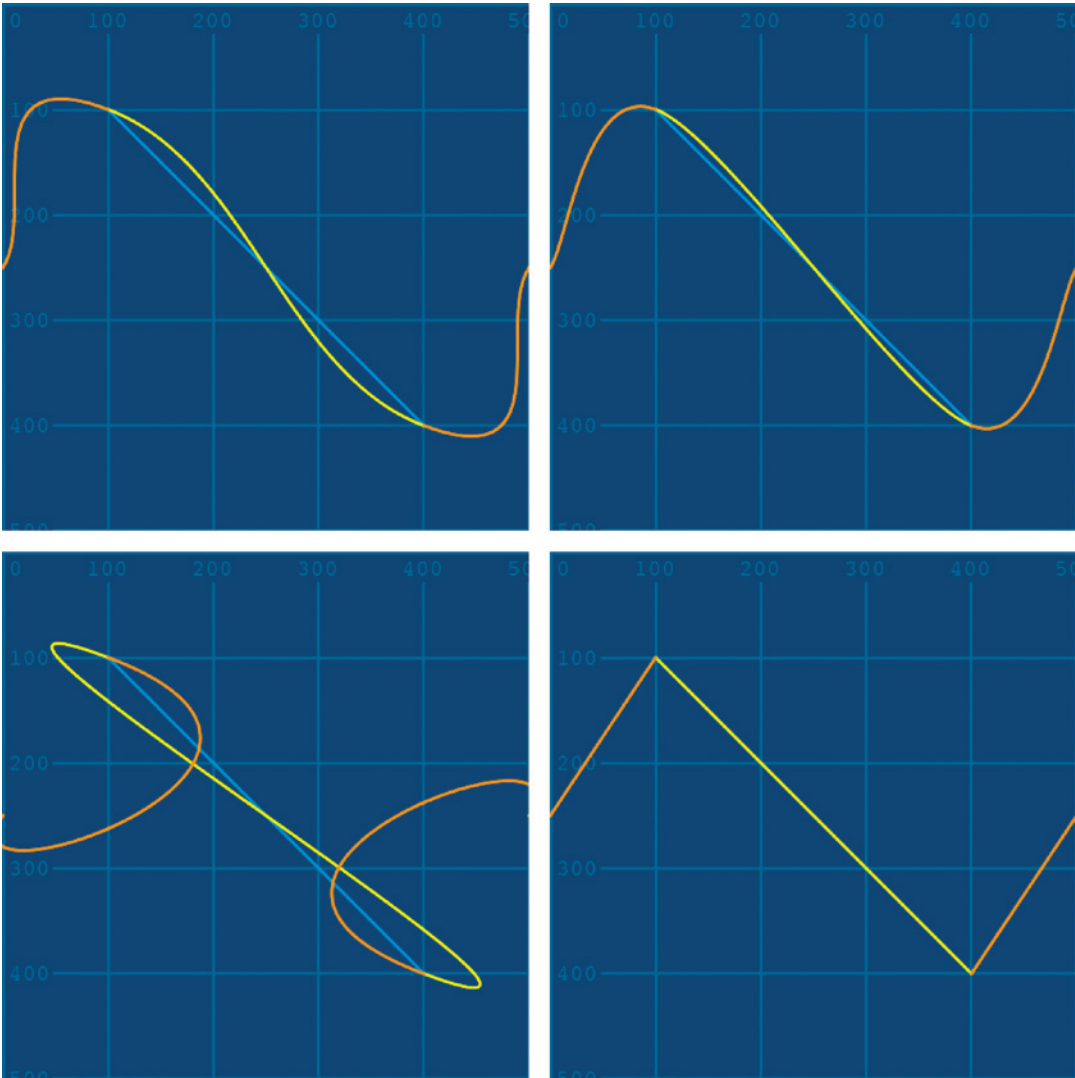
Figure 2-6: Clockwise from the top left: curveTightness(-1), curveTightness(0), curveTightness(1), and curveTightness(5)

The lower right curve in Figure 2-6, with its tightness argument set to 1, fits so rigidly that the result is a straight yellow line. The more you adjust the tightness value away from 1, the more the curve will deform. For curves that overshoot their starting and ending points, use values greater than 1. For instance, at an upper tightness limit of 5 (bottom left), the spline loops as it passes through the starting and ending points. With a tightness argument of -1 (top left), the lengthier spline is rerouted to better align with the points it passes through; hence, there is increased curvature but no looping.

The `curve()` function is intuitive and useful for generating curved lines quickly. However, you're most likely to encounter Bézier curves in 3D modeling, animation, computer-aided design (CAD), and vector illustration software, so let's look at those next.

Drawing Bézier Curves

Bézier curves provide an intuitive and versatile means of modeling smooth curves using a series of anchor and control points. You may have encountered these curves in vector graphics drawing software, such as Adobe Illustrator or Inkscape. In this section, you'll draw curves using the `bezier()` function. In graphics software, you have visual nodes to grab and manipulate; in Processing, you define the positions of your anchor and control points, using `bezier()` function arguments.

Using the bezier() Function

The `bezier()` function takes eight arguments, expanded across multiple lines here for easier readability:

```
bezier(
  anchor_point_1_x, anchor_point_1_y,
  control_point_1_x, control_point_1_y,
  control_point_2_x, control_point_2_y,
  anchor_point_2_x, anchor_point_2_y
)
```

The first and last pair of arguments are the starting and ending points for your curve. When using Bézier curves, you typically refer to the points that your visible lines connect to as *anchor points*. The curvature of the line as it heads away from the first anchor point (*anchor_point_1_x, anchor_point_1_y*) is controlled by the position of its associated control point (*control_point_1_x, control_point_1_y*). The other control point (*control_point_2_x, control_point_2_y*) controls the curvature of the line as it heads toward the ending anchor point (Figure 2-7). This is not spline-like behavior, though; instead, the control points behave more like magnets, causing the line to bulge toward them.

To draw a Bézier curve, create four variables to represent the x-y coordinate pairs of the two control points:

```
. . .
stroke('#FF99FF')  # pink
cp1x = 250
cp1y = 250
cp2x = 250
cp2y = 250
bezier(400,100, cp1x,cp1y, cp2x,cp2y, 100,400)
```
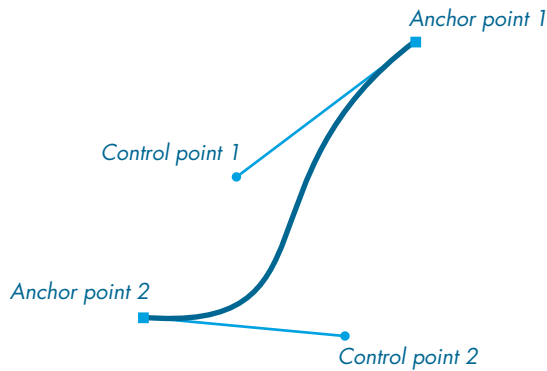
Figure 2-7: The anchor and control points manipulate
the position and curvature of the Bézier curve.

The first pair of bezier() coordinates positions anchor point 1 near the
top right of the grid; the last pair of coordinates positions anchor point 2
near the bottom left. All of the control point variables (cp1x,cp1y, cp2x,cp2y)
reference the center of the display window (250, 250). By placing the control
points along the diagonal path formed between anchor points 1 and 2, you
form a straight line. You'll next shift these control points outward to observe
how this curves the line.

Run this sketch to render a pink line that represents a straightened
Bézier curve (Figure 2-8).
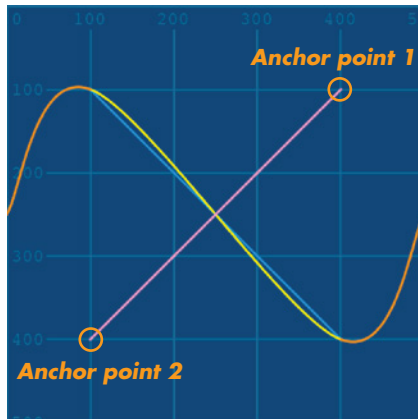
Figure 2-8: The pink line represents a
straightened Bézier curve.

The pink line should cross the yellow curve at the center of the display
window (250, 250).

Positioning Anchor and Control Points

To manipulate the pink line into a curve (Figure 2-9), set the `cp1x` variable to `200`. In addition to this change, add two extra lines of code:

```
. . .
cp1x = 200
. . .
bezier(400,100, cp1x,cp1y, cp2x,cp2y, 100,400)
stroke('#FF0000')  # red
line(400,100, cp1x,cp1y)
```
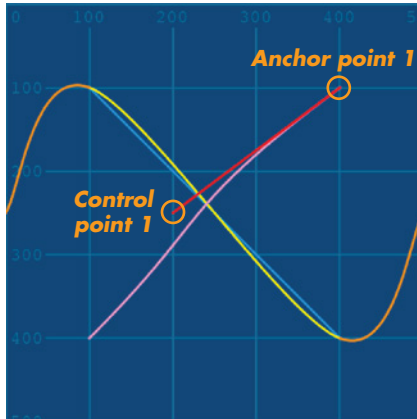

*Figure 2-9: Curving the pink line by
adjusting a control point*

The additional code creates a red line connecting anchor point 1 (`400,
100`) and its control point (`cp1x,cp1y`). This red line is useful because you can
now visualize where the control point sits and which anchor point it controls.
Moreover, sharing variables between the `bezier()` and red `line()` functions
means that each time you adjust the values that position the curve's control
point (`cp1x,cp1y`), the red line adapts accordingly. Setting the value of `cp1x` to
`200` applies curvature to the pink line because—as the control point moves
away from the pink line—the pink line bulges toward it.

The top half of the curve is affected most by the control point that con-
nects to its top anchor point (control point 1); this will become more appar-
ent when you manipulate the control point for the lower anchor point.

Now add another red line to connect (the lower) anchor point 2 and
control point 2:

```
. . .
cp2x = 320
cp2y = 420
. . .
line(400,100, cp1x,cp1y)
line(100,400, cp2x,cp2y)
```

The new red line visually connects anchor point 2 (100,400) to its control point (cp2x,cp2y). Run the sketch to see the result (Figure 2-10). Experiment with different control-point values to see how they affect the curve.
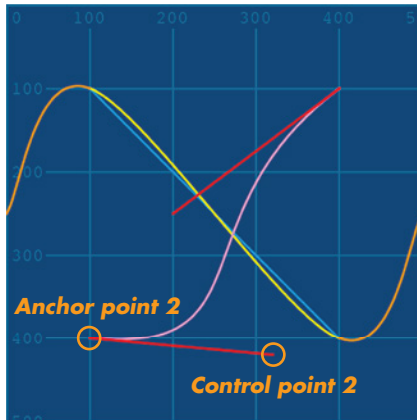
Figure 2-10: Adjusting control point 2

Observe that the lower part of the pink curve is "magnetically" pulled toward control point 2. Knowing where to place the anchor and control points for your desired curve takes some skill. Try downloading and practicing in Inkscape (or Illustrator if you have it installed). Alternatively, try playing The Bézier Game in your web browser at *https://bezier.method.ac/.*

BÉZIER CURVES IN VECTOR GRAPHICS

Vector graphic formats (such as Scalable Vector Graphics, or SVG) employ Bézier curves to render shapes scalable to any size, with no sacrifice in quality. You refer to vector graphics as *resolution independent*, being defined by a series of mathematical formulas rather than a grid of pixels. You can create SVG files in vector graphics software, like Inkscape and Illustrator, by using selectable nodes to position the anchor and control points of Bézier curves (Figure 2-11).
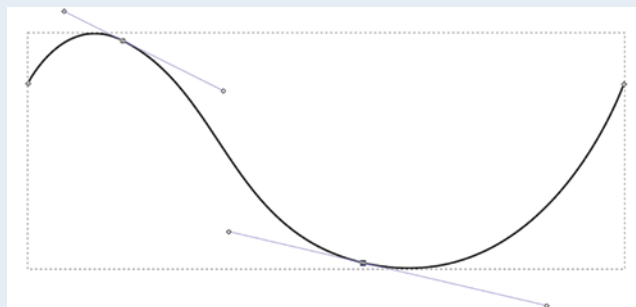
Figure 2-11: Editing a Bézier curve in Inkscape

(continued)

Contrast this with a *raster graphic*, where, as you zoom further and further in toward a given point, discernible squares of color appear (Figure 2-12).
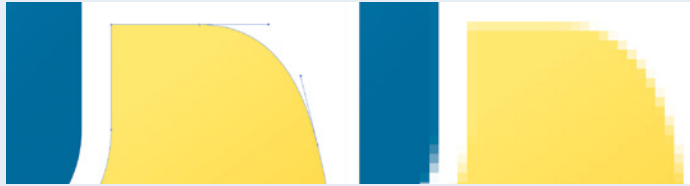
Figure 2-12: Editing a vector version of the Python logo in Illustrator (left), and editing a raster version of the same graphic in Photoshop (right)

This is because pixel-based graphic formats used for photos—such as Joint Photographic Experts Group (JPG) and Portable Network Graphics (PNG)—are composed of a pixel grid, the dimensions of which limit the overall resolution.

You can now draw curved lines by using Catmull-Rom splines and Bézier curves. The curve() and bezier() functions are useful for standalone curves, but to form shapes composed of multiple curve segments, you'll need vertices.

Drawing Shapes Using Vertices

In Processing, a *vertex* is a point used to connect lines in order to form a shape. *Vertices* is the plural of vertex. You can think of vertices as the dots in a connect-the-dots drawing puzzle. For example, a triangle requires 3 vertices; a pentagon requires 5; and a five-pointed star (★) requires 10. When using straight lines and curves to connect vertices, the shape possibilities become limitless. A vertex is not limited to 2D space—for instance, Blender's Suzanne (a monkey head) has around 500 vertices positioned in 3D space (Figure 2-13).
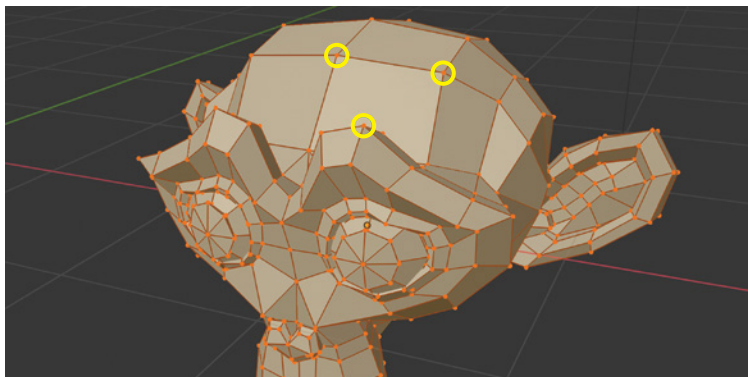
Figure 2-13: Three of the 500 or so vertices circled in yellow

You'll draw a square-type shape by using a series of `vertex()` functions. Create a new sketch and save it as *vertices*. Within the new *vertices* folder, add a *data* folder containing a copy of the *grid.png* file from your preceding sketch (Figure 2-14).

```
vertices                                    ✕

  📁 data
      └ 📄 grid.png
  📄 sketch.properties
  📄 vertices.pyde
```
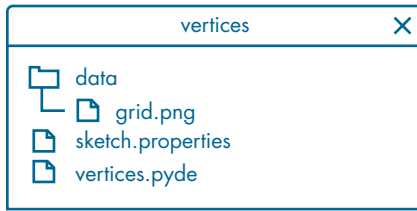
Figure 2-14: The vertices *sketch folder structure*

Add code to set up the initial parameters:

```
size(800, 800)
grid = loadImage('grid.png')
image(grid, 0, 0)
noFill()
stroke('#FFFFFF')
strokeWeight(3)
```

Again, you load and display the grid image to help you gauge coordinates in the display window. Each shape that you draw will have no fill and a white stroke of 3 pixels.

Now, instead of using a `rect()` or `square()` function, use vertices to draw a square:

```
beginShape()  # begins recording vertices for a shape ...
vertex(100, 100)
vertex(200, 100)
vertex(200, 200)
vertex(100, 200)
endShape()    # stops recording
```

The `beginShape()` and `endShape()` functions are essential for separating groups of vertices into individual shapes. Without those two functions, Processing would have to assume that all the vertices in your sketch belong to the same shape. That said, Processing ignores any rogue `vertex()` lines placed outside the `beginShape()` and `endShape()` pair. As depicted in Figure 2-15, the code draws a square with no left side.
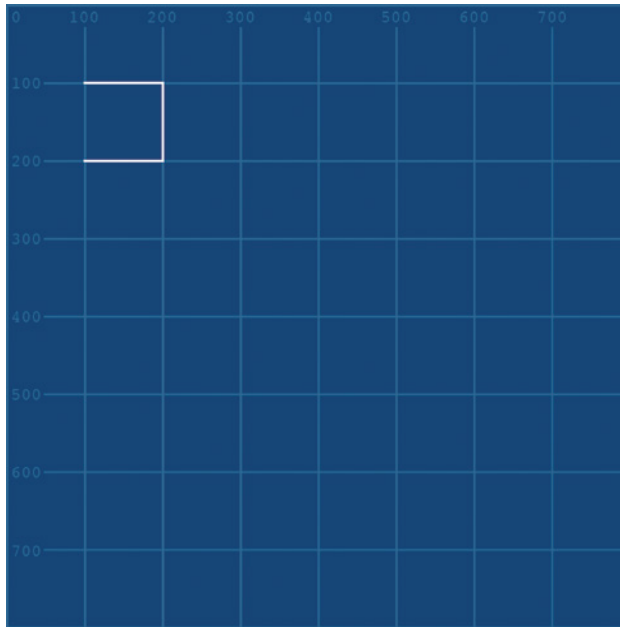
Figure 2-15: An open square drawn using vertices

The shape will not close automatically unless you include an `endShape(CLOSE)` argument or add a final vertex that connects with the start. However, an active `fill()` will fill in color regardless (Figure 2-16).


```
vertices
size(800, 800)
grid = loadImage('grid.png')
image(grid, 0, 0)
noFill()
stroke('#FFFFFF')
strokeWeight(3)

fill('#6633FF')

beginShape()  # begins record
vertex(100, 100)
vertex(200, 100)
vertex(200, 200)
vertex(100, 200)
endShape()     # stops record
```

Figure 2-16: Despite the open side, the shape is filled with color.

You also can provide various parameters to the beginShape() function to determine how the enclosed vertices are connected, if at all (Figure 2-17).
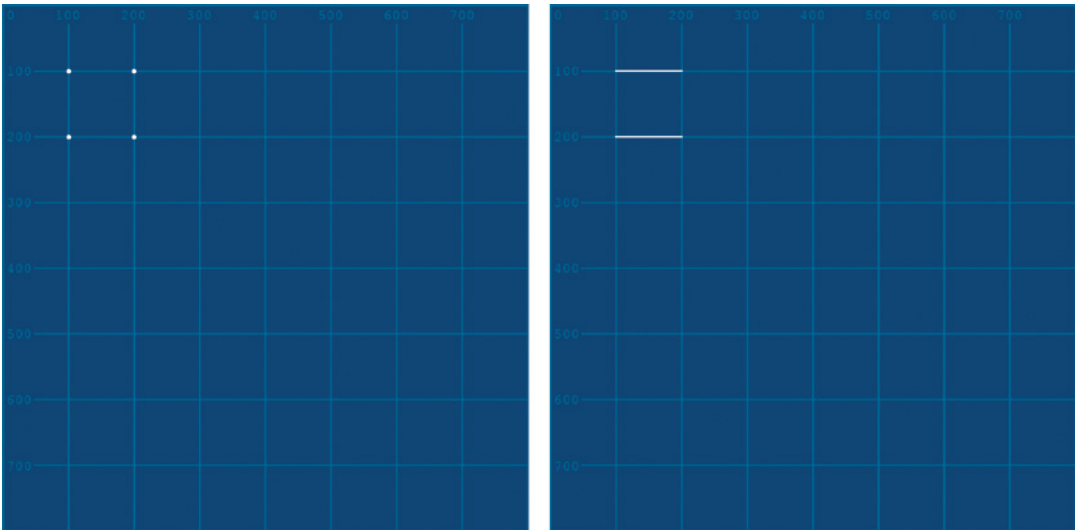
Figure 2-17: The functions beginShape(POINTS) (left), and beginShape(LINES) (right)

For a shape composed of only dots, use beginShape(POINTS). For a line between every other vertex, use beginShape(LINES). Consult the reference for more details on beginShape() arguments.

Bézier Vertices

The bezierVertex() function allows you to draw curved lines between vertices. A curveVertex() function is also available for Catmull-Rom-type curves, but this book focuses on the Bézier type, as it provides for greater control and more graceful curves.

The bezierVertex() function takes six arguments. To understand how those arguments operate, you'll work toward completing the remaining shapes shown in Figure 2-18.

I have manually added the pale blue lines, the dotted tips of which provide a visual indication of the control points. Use these lines for reference purposes only; you don't need to redraw them.
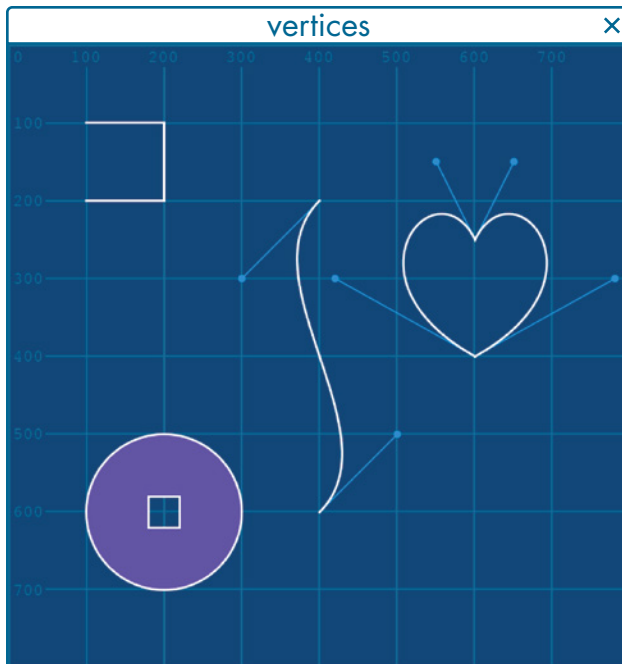
Figure 2-18: A Chinese coin (lower left), S-curve (middle), and heart (right)

S-Curve

The *S-curve* is just a curved line that comprises two vertices, with each vertex attached to its own control point. You'll draw it with a `bezierVertex()` function to keep this first example as simple as possible, but ordinarily, you would draw an S-curve by using `bezier()`.

Within `beginShape()` and `endShape()`, combine the `bezierVertex()` and `vertex()` functions however necessary. Your first point, however, is always created with `vertex()`. Begin a new shape and plot the first (in this case, upper) vertex:

```
. . .
# s-curve
beginShape()
vertex(400, 200)  # starting (upper) vertex
endShape()
```

Run the sketch. There is no second vertex with which to form a line, so the isolated vertex should appear as a point at (400, 200).

Now add the second vertex by using `bezierVertex()`:

```
. . .
# s-curve
beginShape()
```

```
vertex(400,200)  # starting (upper) vertex
bezierVertex(
  300, 300,  # control point for the starting vertex
  500, 500,  # control point for the second (lower) vertex
  400, 600   # second (lower) vertex coordinates
)
endShape()
```

The last pair of `bezierVertex()` arguments (`400, 600`) denotes the position of the second (lower) vertex. The second vertex is attached to a control point positioned by the second pair of arguments (`500, 500`). The first pair of arguments (`300, 300`) represents the control point for the `vertex()` function that immediately precedes `bezierVertex()`. With the positions of the vertices presented for you in the reference image (Figure 2-18), creating this shape (Figure 2-19) is really just a matter of typing in the correct sequence of coordinates.
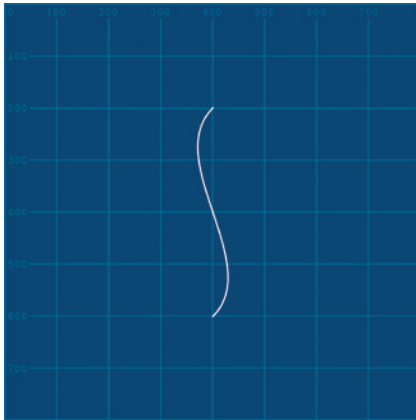
Figure 2-19: The complete S-curve

This is an open shape, so it would look odd if filled. Next, you'll examine a closed shape, but feel free to experiment with different vertex and control-point values before moving along.

Heart

You can think of the heart shape as two curved lines connected to two vertices. To begin, draw one half of the heart (Figure 2-20):

```
. . .
# heart
beginShape()
vertex(600, 400)
bezierVertex(420,300, 550,150, 600,250)
endShape()
```
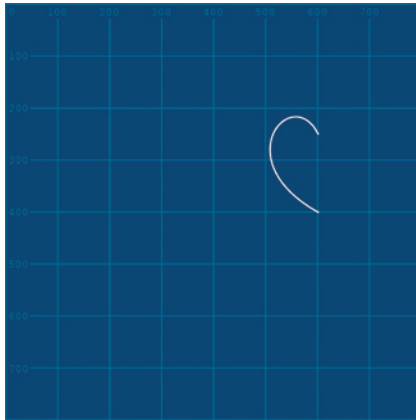
Figure 2-20: Half a heart

All that is left for you to do is complete the right half of the heart. Add a second bezierVertex() line and see if you can fill in the missing arguments:

```
. . .
# heart
beginShape()
vertex(600, 400)
bezierVertex(420,300, 550,150, 600,250)
bezierVertex(___,___, ___,___, 600,400)
endShape()
```

Refer back to Figure 2-18 to see where the control points lie. Remember that you can access all of the solutions to the challenges at *https://github.com/ tabreturn/processing.py-book/*.

Chinese Coin

Round metal coins with square holes in the center were first introduced in China many centuries ago, but replicating that shape makes for a good example to learn Processing. To create the purple coin shape in Figure 2-18, you'll use the beginContour() and endContour() functions to subtract a square from a circle.

First, you'll create the outer shape by using the beginShape(), endShape(), and vertex() functions. You'll then place the beginContour() and endContour() functions within the beginShape() and endShape() functions. Within this pair of contour functions, you'll draw a second shape that's also composed of vertex() and bezierVertex() functions; the contour functions subtract this shape from the outer shape.

The first challenge is creating the outer circle. The `beginContour()` and `endContour()` functions cannot subtract from predefined shape functions—like `rect()`, `ellipse()`, or `circle()`—so you need to construct the outer circle by using vertices. However, it is possible to draw circles by using Bézier curves, which you'll do by creating a diamond and then using the control points to form it into something round.

Begin by forming a diamond shape with `vertex()` functions (shown in Figure 2-21):

```
. . .
# coin
beginShape()
vertex(100, 600)
vertex(200, 500)
vertex(300, 600)
vertex(200, 700)
vertex(100, 600)
endShape()
```
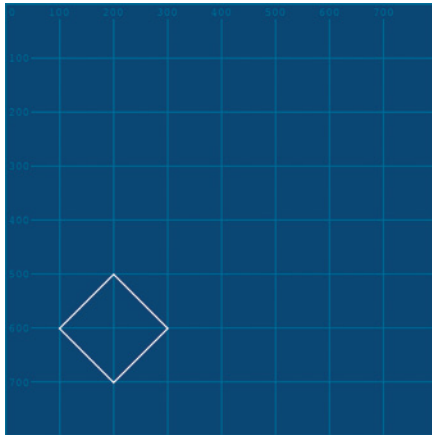

Figure 2-21: The diamond shape that you'll form into a circle

With the vertices in the correct positions, you can proceed to add curvature to the diamond. Of course, this will require `bezierVertex()` functions, for which you'll reference the coordinates of the vertices currently in place. For an idea of where to position the additional control-point coordinates, see Figure 2-22.
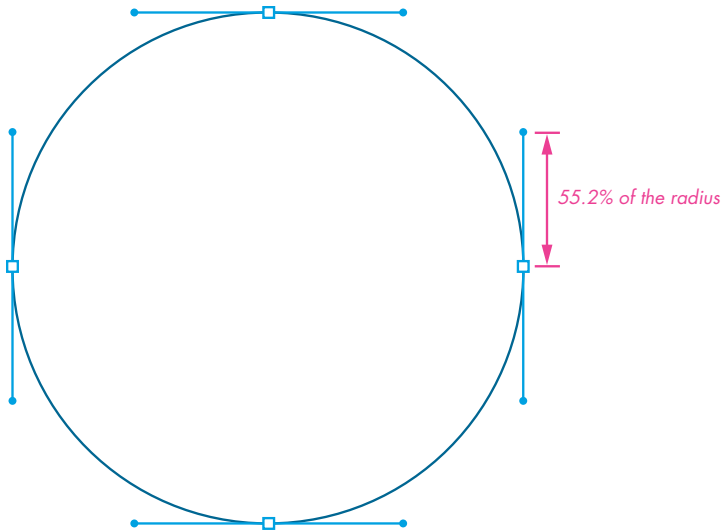
Figure 2-22: Positioning vertices and control points to form a circle

Figure 2-22 indicates how the control points should be positioned to form the most circular shape. Now replace each vertex() with a bezierVertex() function. Remember, though, that the first point must remain a vertex() to form your circle (Figure 2-23):

```
# coin
beginShape()
vertex(100, 600)
bezierVertex(100,545, 145,500, 200,500)
bezierVertex(255,500, 300,545, 300,600)
bezierVertex(300,655, 255,700, 200,700)
bezierVertex(145,700, 100,655, 100,600)
endShape()
```
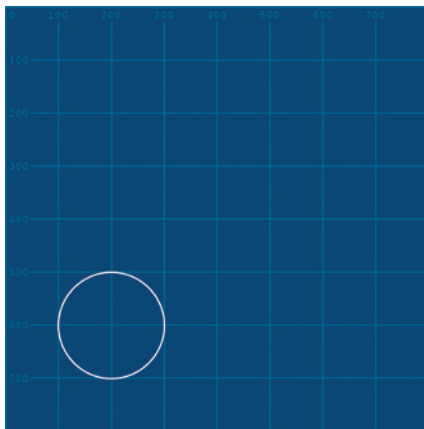

Figure 2-23: A circle formed using
bezierVertex() functions

With the circle in place, you can go about removing a square from the middle. Once again, define this square by using vertices and not a predefined shape function, like rect() or square(). This is a relatively straightforward exercise, but be aware that you need to use *reverse winding* for the subtracted shape: you must lay the vertices of the square in a direction that's opposite to the one you used to place the vertices of the exterior shape (the circle).

Read through the circle code again and notice that the vertices are plotted in a clockwise sequence; this means that the square's vertices must be plotted counterclockwise—that is, opposite to the winding of the shape from which it will subtract. If you fail to get this direction correct, no subtraction will take place.

Place the square's vertices within a beginContour() and endContour() function. Of course, you can't observe the effect (shown in Figure 2-24) unless you add a fill:

```
  # coin
❶ fill('#6633FF')
  beginShape()
  vertex(100, 600)
  bezierVertex(100,545, 145,500, 200,500)
  bezierVertex(255,500, 300,545, 300,600)
  bezierVertex(300,655, 255,700, 200,700)
  bezierVertex(145,700, 100,655, 100,600)
❷ beginContour()
  vertex(180, 580)
  vertex(180, 620)
  vertex(220, 620)
  vertex(220, 580)
❸ endContour()
  endShape()
```
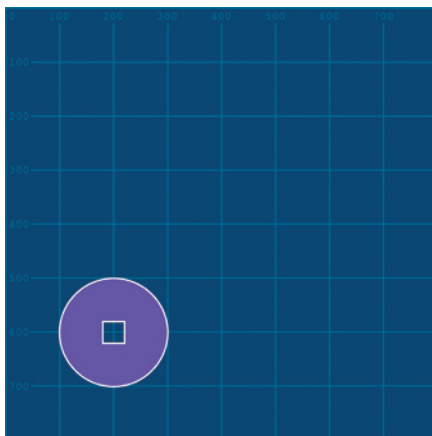

Figure 2-24: The completed coin

Without the fill ❶, you would see only white outlines. The `beginContour()` function ❷ starts recording the vertices that make up the negative shape. No `bezierVertex()` functions are necessary, because a square has no curves. The vertices follow a counterclockwise sequence, beginning at the upper left corner of the square (`180, 580`), proceeding directly downward (to `180, 620`) and then farther around before the `endContour()` stops recording ❸.

Using Vector Graphics Software for Generating Shapes

You can use vector graphics drawing software to draw shapes, and then reference the positions of the vertices and control points for writing Processing code. This is how I mapped out the blue guidelines for the Python logo shown in Figure 2-25.
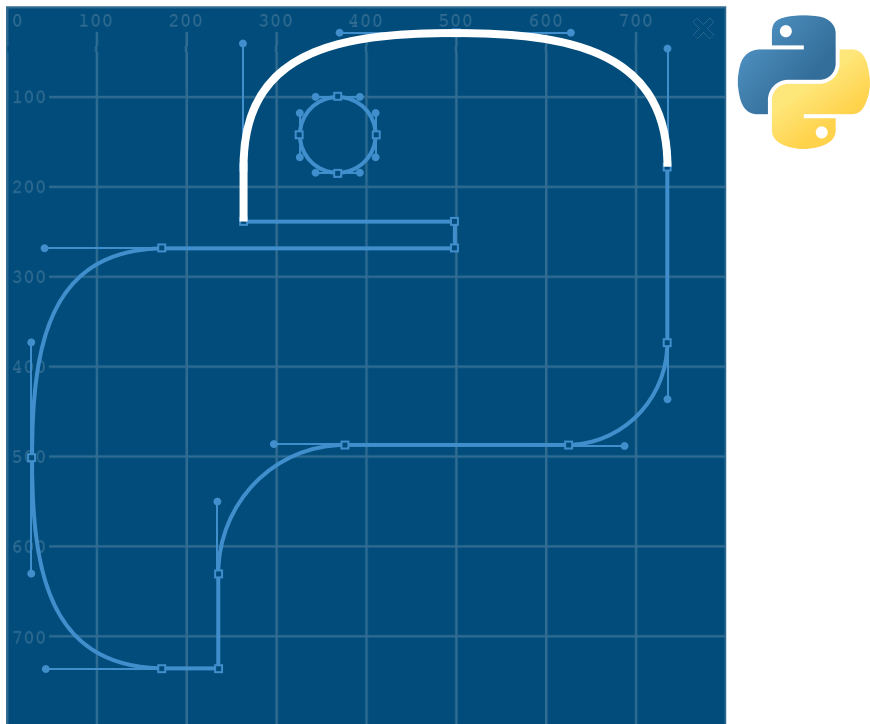
Figure 2-25: Tracing a Python logo that includes the positions of the vertices and control points. (The Python Software Foundation logo trademark policy is available at https://www.python.org/psf/trademarks/.)

If you're up for a challenge, clear out your *curves* sketch and try finishing the half of the Python logo I've begun in Figure 2-25. Here is some code to get the outline started:

```
beginShape()
vertex(262, 238)
vertex(262, 178)
```

```
bezierVertex(262,40, 370,30, 500,30)
bezierVertex(630,30, 730,40, 735,178)
endShape()
```

You can also export vector graphics as SVG files for use in Processing with the loadShape() and shape() functions, as opposed to the loadImage() and image() functions. But be warned: SVG support is not always dependable, and you may spend some time fiddling with your SVG export settings to get them to display properly in Processing.
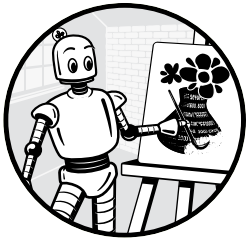
Summary

You've now learned most of Processing's essential drawing features. Using a grid graphic as a reference for your coordinates, you learned to plot curves that mimic physical splines. In addition, you learned to draw Bézier curves—smooth, graceful curves that you can control with anchor and control points. You also saw how to draw shapes by using a series of vertices. When you connect vertices with straight lines and curves, the shape possibilities are limitless. You'll be using curves, vertices, and the skills you learned in Chapter 1 in many of the tasks to come.

In Chapter 3, you'll move on to explore Processing's text features. This includes drawing text to the display window, styling it, and loading fonts. You'll also look at Python's built-in features for manipulating string data. Later in this book, you'll use text functions to label graphs and graphical interface elements, and to add speech bubbles to images.

3

INTRODUCTION TO STRINGS AND WORKING WITH TEXT

In Chapter 1, you created a 'Hello, World!' string and printed it to the console, but Python can do far more than just print string data. In this chapter, you'll use operators, functions, and methods to manipulate strings. Strings are fundamental data types common to most programming languages, and you'll use them in almost all the programs you write. If you need to communicate information to your user, capture input from text fields, retrieve data from the web, or perform just about any task that involves text, you'll be using strings.

In this chapter, you'll also learn how to use Processing's text functions to render any string as text in the display window. Processing can draw text in various colors and styles, using different fonts, at different sizes and positions. You might use these features to paint with letters, label a graph, display a table of high scores, or construct an interactive interface.

Strings

Before exploring Processing's text-rendering functions, you'll need a proper introduction to strings. By definition, a *string* contains a sequence of one or many characters. For example, 'hello' is a string that's five characters long; it begins with an *h* and ends with an *o*. You already briefly encountered the string data type in Chapter 1, where you used it to define hexadecimal color values and print text messages to yourself in the console.

To create a new 'hello' string and assign it to a variable named `greeting`, use the following code:

```
greeting = 'hello'
```

Python recognizes `hello` as a string because it's wrapped in quotes. You can use single or double quotes, but always make sure you close them using the same type with which you opened them.

In Python, you can manipulate strings in various ways. To convert `hello` to `Hello!`, you would make the first character uppercase and insert an exclamation mark at the end of the string. Python has many built-in features for performing those types of operations, and I cover some of the most useful features in this section.

You'll look at how to combine strings, and how to find, count, and extract specific sequences of characters. Most of those features work exclusively with the string data type. For instance, you cannot convert an integer or a floating-point value to uppercase, because those things are numbers. And if you tack an ! character to the end of a number, it's not a number anymore; it's a string with digit characters and an exclamation mark. On the other hand, you can't divide a string by a number.

The following example uses the division operator to divide the integer 6 by 3, which prints a 2 in the console. But attempting to divide 'hello' by 3 results in an error:

```
print(6 / 3)       # displays a 2
print('hello' / 3) # displays an error
```

Python cannot divide a string by an integer, so you get a `TypeError` message. However, certain mathematical operators do work on strings. For instance, `'hello' * 3` gives you `hellohellohello`. Later in this chapter, you'll learn how to use the + operator to join strings.

Creating Strings in Python

Let's begin by creating a few new string variables, looking at the way Python deals with different kinds of quotation marks, and working around some issues you might encounter when creating strings.

Start a new sketch and save it as *strings*. Add the following code:

```
greeting = 'Hello, World!'
print(greeting)
```

When you run the sketch, the print() function writes Hello, World! to the console.

Recall that Python expects a string to begin and end with quotation marks, so what happens when the string itself contains a quote character? Add another string variable to see what happens when you have unpaired quotes:

```
. . .
whatsup = 'What's up?'
```

Python interprets this string as What, ignoring everything after the apostrophe. Some dangling characters and an unpaired quote (s up?') are left over. Run the sketch and observe the error message (Figure 3-1).
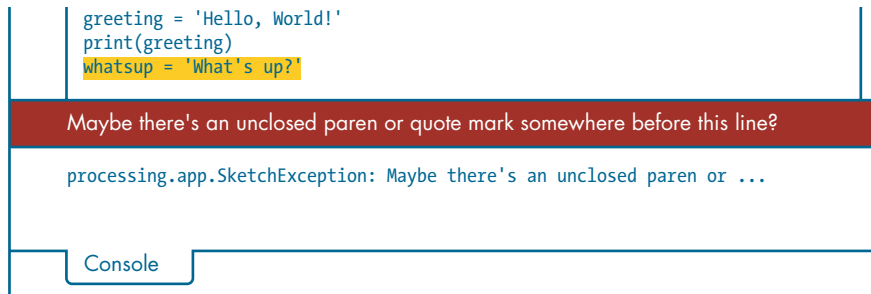
Figure 3-1: An error caused by an apostrophe

To fix this, use double quotes:

```
whatsup = "What's up?"
```

Or, you can *escape* the apostrophe character by using a backslash:

```
whatsup = 'What\'s up?'
```

The backslash indicates that Python should treat the apostrophe as an ordinary character, not part of the language syntax. If you print the whatsup variable now, it displays this:

```
What's up?
```

Note that no backslash displays in the console output.

The backslash is an escape character, so if you need to include a backslash in your string, you must prepend it with another backslash. For example, print('\\') displays a single backslash in the console.

You've seen how to nest a single quote within a string delimited in double quotes. This works both ways, though. For example, add a new question variable that uses double quotes nested within single quotes:

```
greeting = 'Hello, World!'
print(greeting)
```

```
whatsup = "What's up?"
question = 'Is your name really "World"?'
print(whatsup)
print(question)
```

Run the sketch to confirm that it has no errors. The console should display the contents of the three `print` statements.

Using Concatenation and String Formatting

The + operator performs arithmetic addition on integers and floats, but you also can use the + operator to concatenate multiple strings into a series or chain. *Concatenation* is programming terminology for *joining together*, and it's useful for many tasks, such as chaining together words into sentences and paragraphs. Try this example in your sketch:

```
. . .
all = greeting + whatsup + question
print(all)
```

This should display the following line in the console:

```
Hello, World!What's up?Is your name really "World"?
```

Note that concatenation joins strings together precisely as they are defined, with no additional spaces, so you need to insert the required space characters explicitly.

To fix the preceding output, edit the `all` variable line:

```
. . .
all = greeting + ' ' + whatsup + ' ' + question
print(all)
```

The console should display the following:

```
Hello, World! What's up? Is your name really "World"?
```

The line now includes the spaces specified in the code.

An alternative to concatenation is *string formatting*, and Python provides the `format()` method for this (I explain more about methods in "String Methods" on page 60). What you need to understand here is that `format()` works by substituting placeholder symbols with values, as opposed to chaining them together in a sequence. You'll find that the concatenation operator is okay for simpler tasks, but it can be clumsy when you're constructing lengthier and more intricate strings.

Here's the same line constructed using `format()`:

```
. . .
all = '{} {} {}'.format(greeting, whatsup, question)
print(all)
```

In this approach, Python substitutes each pair of curly brackets ({}) with its corresponding variable—that is, `greeting` for the first pair of curly brackets, `whatsup` for the second, and `question` for the third. This saves you from needing to insert each space character by using `+ ' ' +`.

If the `format()` alternative doesn't seem much simpler, consider this example that uses concatenation:

```
firstname = 'World'
o2 = 21
hi = "Hi! I'm " + firstname + ". My atmosphere is " + str(o2) + "% oxygen."
```

If you print `hi`, you get `Hi! I'm World. My atmosphere is 21% oxygen.` With concatenation, you have to place your space characters carefully, and it's tricky to read what the `hi` line is doing. Moreover, you have to wrap the `o2` variable in the `str()` function to convert the value from a number to a string; if you don't, Python will attempt (and fail) to add an integer and a string arithmetically.

Compare this to using the `format()` approach for the same line:

```
hi = "Hi! I'm {}. My atmosphere is {}% oxygen.".format(firstname, o2)
```

This provides a better idea of the result you're going to get. The `format()` method also manages the conversion of numbers to strings. Use whichever approach works best for the task at hand.

Working with String Length

The `len()` function returns the total number of characters in a string. You might use it to check whether a string contains more than 1 character or to verify that it fits into a tweet (280 characters). You can also use the `len()` function to find the total number of items in a list (Chapter 7) or dictionary (Chapter 8).

The `len()` function accepts a single argument; try this using your `greeting` variable:

```
print(len(greeting))
```

This should display 13, the total number of characters in the `greeting` string.

So far, you've learned how to define new strings and construct strings from smaller strings, but you can do far more in Python. In the next section, you'll learn how to manipulate strings by using slice notation and string methods.

String Manipulation

Let's add code to your working sketch so you can try some string manipulation methods. You'll extract partial strings by using slice notation, convert between uppercase and lowercase characters, and find and count the

occurrences of specific character sequences. You can use these techniques to automate the processes involved in handling string data—for example, to scan data for keywords, dissect strings, or shorten them. Feel free to experiment with values and arguments on your own to see how things respond.

Slice Notation

Python *slice notation* provides a simple yet powerful means of extracting characters from strings. You can use slice notation to retrieve a single character or substring. A *substring* is any contiguous sequence of characters that form part of a longer string. For example, a string that's a URL might begin with the substring `http://`.

To experiment with slice notation on a new string variable named `url`, add this variable to your *strings* sketch:

```
url = 'http://www.nostarch.com'
```

You'll specify the position (*index*) of the character(s) you want to retrieve by using a pair of square brackets (`[]`). To keep things simple, let's extract the first character in the `url` string. Note, however, that this indexing system is zero-based, meaning that the character indices start at 0, not 1. See Figure 3-2 as a reference for the character indices.
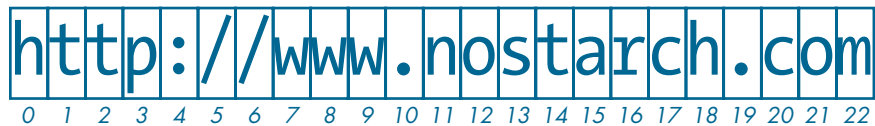
Figure 3-2: The string indexing system begins at 0.

Use `0` (zero) to retrieve the first character:

```
print(url[0])      # displays: h
```

The console should display an `h`.

The index for the second character is `1`:

```
print(url[1])      # displays: t
```

The console should display a `t`, the first `t` in `http`.

Use a colon (`:`) to specify a range of characters. Use this to extract the *scheme* (`http`) along with the colon-slash-slash (`://`) in the URL string, which spans from index `0` up to index `7`:

```
print(url[0:7])    # displays: http://
```

The `:7` to the right of the `0` retrieves the characters up to, but not including, the first `w`. But because your range begins at index `0`, you can omit the `0` in front of the colon to produce the same result:

```
print(url[:7])     # displays: http://
```

The colon precedes the index value (7), which means that Python must retrieve everything from the left/start of the string up to the seventh character.

If you place the colon after the index, Python returns everything from the specified index to the end of the string. You can use this to retrieve everything to the right of the colon-slash-slash, which is the part of the URL you usually type into the browser address bar:

```
print(url[7:])     # displays: www.nostarch.com
```

You now should see www.nostarch.com in your console. This is a combination of the URL's *subdomain* (www), *domain* (nostarch), and *top-level* domain (com), separated by dot characters (Figure 3-3).

Figure 3-3: The parts of a URL

You can isolate each part of the URL with string-slicing operations. Assuming that the top-level domain (com) is always three characters, you can retrieve it by using an index of -3 followed by a colon:

```
print(url[-3:])     # displays: com
```

A negative value counts index positions starting from the end (right side) of a string, so url[-3] will retrieve just c. You can include a colon to retrieve the c and every character that follows it. No matter how long the URL, this code will always display the last three characters. Conversely, using [:-3], with the colon to the left of -3, retrieves everything up to the third-to-last character (http://www.nostarch.).

To extract the domain (nostarch), retrieve the substring between index 11 and -4:

```
print(url[11:-4])  # displays: nostarch
```

This will adapt to any domain. For instance, if you change the url value to http://www.nostarchpress.com, Python prints nostarchpress. But this works only if the scheme is *http* and the subdomain is *www*. You can use string methods that will adapt to schemes, subdomains, and top-level domains of any length.

This notation slices strings in a few other ways, but these should be sufficient for now. You can also use slice notation to extract items from lists and dictionaries, so you'll encounter it again in the chapters that deal with those data types.

String Methods

String methods perform various operations on strings, such as converting characters between uppercase and lowercase, and searching for and counting characters and substrings. You'll use string methods in your sketch to verify that your URL contains a scheme, subdomain, domain, and top-level domain. This is not an exhaustive review of string methods, but it will familiarize you with how some of them operate. Any decent Python reference will cover the rest.

Methods vs. Functions

A Python *method* looks and behaves much like a function. You call a function by its name—like print()—and it performs a predefined task for you. Methods work similarly, but they're associated with specific objects, such as strings for string methods. A function may or may not accept arguments, depending on the function you're using; the same is true for methods.

As an example, let's contrast the len() *function* with a *method*. There's no len() method, but we'll pretend there is to focus on the syntactical differences between how you write a method versus a function.

Recall that the len() function returns the total number of characters in any string:

```
urllength = len(url)
```

The len() function takes the url argument and returns the length of the string it holds. The total length of the url string is 23 characters, so the variable urllength is equal to the integer 23.

Methods begin with a dot (.) and are appended to data you want to affect. If the len() function were a method, you would write it like this:

```
urllength = url.len()
```

Next, you'll use the upper() method to convert string characters to uppercase.

upper() and lower() Methods

The upper() method returns a version of the string with all the lowercase characters converted to uppercase. It takes no arguments. Here's an example:

```
urlupper = url.upper()
print(urlupper)            # HTTP://WWW.NOSTARCH.COM
```

The upper() method is a string method, so you must append it to a string. The syntax might look similar to format(), which is the method you used to replace curly brackets with text values in strings earlier. In this instance, the variable urlupper is equal to HTTP://WWW.NOSTARCH.COM. This method might be

useful to emphasize certain key phrases when you're unable to use bold or italics. The lower() method is the inverse of upper(), and it converts all upper-case characters to lowercase.

count() Method

Now, let's verify that the url string contains a *www* subdomain. The count() method returns the total number of times that a character, or character sequence, appears in a string, and it needs an argument to indicate which character(s) you want to count. For instance, you can use the count() method to verify that the URL contains three instances of the letter *w*:

```
print(url.count('w'))    # 3
```

Your console should confirm that there are three *w* characters. But it doesn't indicate whether they are contiguous; that letter might be scattered throughout the string. To be more explicit, use an argument of 'www':

```
print(url.count('www'))  # 1
```

The substring www appears only once in this string. Still, you can't be sure that this is the subdomain. What if the domain part of the URL has a *www* in it? You could be more specific and count the instances of http://www, but HTTP isn't the only scheme for web addresses. For example, *HTTPS*, a secure extension of HTTP, is used to encrypt communication over computer networks. To make matters more complicated, a subdomain can be something other than *www*.

find() Method

Let's try another approach. The find() method returns the index of any character or substring. Note how the colon-slash-slash (://) splits the scheme and subdomain. Use the find() method to retrieve the index of the colon-slash-slash. Add code to find the index, store it in a variable named css, and then use this to extract the scheme:

```
css = url.find('://')    # 4
scheme = url[:css]       # http
```

The find() method retrieves the index for the first occurrence of any :// in the url string. More specifically, it's the index of the first character in the substring, the colon. If the substring cannot be found, the result is a -1. In this instance, it's an index of 4. Note that this argument is case-sensitive.

The subdomain sits between the colon-slash-slash and the first dot. Use the find() method to locate the index of the first dot, and use slice notation to extract and assign the subdomain to a variable named subdomain:

```
dot1 = url.find('.')         # 10
subdomain = url[css+3:dot1]  # www
```

The css+3 is equal to 7, the index of the first *w* in *www*. I've added the 3 to offset the starting index by the length of the colon-slash-slash. This will work for *www* or any other subdomain (although you will encounter issues if there's no subdomain).

The top-level domain (*com*) spans from the second dot to the end of the string. If a character or substring appears multiple times—like the dot—you can provide a second find() argument indicating the index where the search should begin. You can use the dot1 variable for this offset, but you need to add 1 to start from the character immediately after it. Assign the top-level domain to a variable named tld:

```
dot2 = url.find('.', dot1+1)   # 19
tld = url[dot2 + 1:]           # com
```

The dot2 variable is equal to 19, the index of the second dot in your URL. In the tld line, I've added 1 to the start index argument (of 19), because I don't want the dot in *.com*.

The find() method can accept an additional third argument to indicate where along the string the search should terminate.

Finally, assign the domain (*nostarch*) to a variable named domain:

```
domain = url[dot1+1:dot2]      # nostarch
```

The domain substring sits between the first and second dot, but add 1 to dot1 to avoid retrieving the first dot character.

You've now separated a URL into parts by using slice notation. Combining slice notation with string methods provides a more robust way of doing this, so your program can handle schemes, subdomains, and top-level domains of varying lengths.

NOTE *For more powerful and dynamic find operations, you can use* regular expressions *(also known as* regex*). Regular expressions are a popular way to define search patterns using characters. Python and many other programming languages include support for regex, but this book does not cover that topic.*

In the next section, you'll learn how to use Processing text functions to display strings as text in the display window, so you're no longer constrained to printing strings in the console. You can use text decoratively, to label elements in your visual output, or provide feedback to users.

Typography

Typography refers to the arranging and styling of text (or *type*) to make it more readable and aesthetically appealing. Typographical treatment can truly make or break a design. For instance, headings work best if they stand out from the rest of your text; letter spacing should be tighter than word spacing, and you probably agree that cursive fonts are not ideal for road signs. Although I wouldn't recommend that you lay out a book in Processing, it does offer useful functions for controlling the appearance of text.

Fonts

Fonts comprise many glyphs; a *glyph* is any individual character, such as *A*, *a*, or *?*. If you don't specify which font Processing should use to draw text, it relies on a predefined default. Your computer includes a bundle of pre-installed fonts, but the selection varies among operating systems. You can also install additional fonts on your system to expand your selection. However, you might run into problems if you're moving or sharing sketches between computers (with different collections of fonts). If a sketch requires a specific font, and it's not installed, Processing cannot load it. To avoid these issues, I'll explain how to bundle font files with your sketches.

Because early computer fonts were pixel-based, they required a separate set of glyphs for each font size. For example, if a font had three sizes and an italic variant, it included six complete sets of character graphics. However, modern fonts are vector-based, which is why you can scale text to any size you like without encountering pixelation. You no longer require a file for every font size, but bold and italic variants are still separate font files.

By default, Processing will render text in the display window by using a standard *sans serif* font. In font terminology, *serifs* are the small lines attached to the tips of characters (circled in Figure 3-4). The term *sans* means *without*; hence, a sans serif font has no serifs.

serif

sans-serif

monospace

Figure 3-4: Classifying fonts

Monospace fonts may also be serifed, but what distinguishes them is that each character occupies the same amount of horizontal space. *Proportionately spaced* fonts (like the serif and sans serif examples in Figure 3-4) make type more legible by using built-in metrics that specify how far a given character should sit from its neighbors. For example, having an *i* and *m* character occupy the same size "container" results in awkward spacing issues, which many monospaced fonts attempt to resolve by adding oversize serifs to the *i* and cramping the *m* (Figure 3-5). This also means that monospace characters vertically align across multiple lines of text.

mmm

mmm

iii

iii

Figure 3-5: Monospace characters have a fixed width

That said, monospace fonts are more legible in certain situations. For instance, a monospace font is useful when you need to have characters line up in columns:

```
Sam  Jan  Amy  Tim | Total
99   359  11   3   | 472
```

This characteristic makes monospace fonts preferable for writing code, which is why the default font for the Processing editor (and every other code editor) is monospaced.

Text Functions

Let's create a new sketch to experiment with Processing text functions. You'll use these functions to draw text in the display window and to set your font, font size, line spacing, and text alignment.

Start a new sketch and save it as *typography*. Add the following code to get started:

```
size(500, 320)
background('#004477')
fill('#FFFFFF')
stroke('#0099FF')
strokeWeight(3)
```

This code sets the background to blue and the fill color to white. As you'll soon see, the fill() color will affect the text you draw. Any strokes are pale blue and 3 pixels wide.

A *pangram* is a sentence that uses every letter in a given alphabet at least once. Create a variable called pangram that holds a perfect English pangram:

```
pangram = 'Quartz jock vends BMW glyph fix'
```

From here on, you'll render different versions of the string stored in pangram, as shown in Figure 3-6.

To recreate Figure 3-6, begin with the text() function, which draws text to the display window, the font color of which is determined by the active fill:

```
text(pangram, 0, 50)
```

Figure 3-6: You will render these versions of the same pangram.

Run the sketch. You should see the first (top) version of the pangram rendered in the display window. The arguments (`pangram, 0, 50`) represent the string value, x-coordinate, and y-coordinate, respectively. You can add additional third and fourth arguments to specify a width and height for the text area, which you'll use shortly.

The `textSize()` function sets the font size (in pixels) for all subsequent `text()` functions. Add the following code to display the second version of the pangram:

```
textSize(20)
text(pangram, 0, 100)
```

Run the sketch to confirm that you have a smaller and larger version of the pangram.

Observe that the vertical, pale blue line (Figure 3-6) precisely marks the end of the longest/larger line of text. The purpose of adding this line is to explore the `textWidth()` function, which you use to calculate the width of any text you might display. In this instance, you want to measure the width of the second pangram and draw a vertical line at the end of it. Use `textWidth()` functions as arguments for a line function:

```
line(
  textWidth(pangram), 0,
  textWidth(pangram), height
)
```

The width of the pangram now serves as the starting as well as ending x-coordinate for the line; the starting and ending y-coordinates are the top and bottom edges of the display window, respectively. This will draw a pale blue vertical rule, the height of the display window, that marks the end of the second pangram.

You'll render the third pangram in a serif font. To switch to a different font, you need to know the font name to reference. To list the fonts installed on your computer, use `PFont.list()`:

```
print(PFont.list())
```

Scroll through the console output to see if you can spot `Cambria` or `Georgia`. Both are serif fonts. If neither Cambria nor Georgia is installed on your system, you won't find them in the list. In that case, any other serif font will work, such as Times New Roman.

Processing uses its own font format, so you need to convert your font before you can use it, using the `createFont()` function.

Add a `createFont()` line that includes a string argument with the name of the serif font you will use:

```
seriffont = createFont('Cambria', 20)
```

The `createFont()` function takes two arguments: a font name (as it appears in the console listing) and point size. The preceding line assigns the converted font to a variable named `seriffont`, which you'll use in the next step.

LOADING FONT FILES DIRECTLY

The fonts listed by `PFont.list()` reside somewhere on your computer, but the location varies among systems. If you know how to locate these files, it's a good idea to place any you use—TrueType Font (TTF) or OpenType Font (OTF)—in your sketch's *data* folder, because not every computer is likely to have the fonts you've used installed, or perhaps you'll need to reopen this sketch sometime in the future on a freshly installed system. If you've downloaded font files from the web, place a copy of them directly in the *data* folder.

To load fonts directly from the sketch's *data* subfolder, reference the full filename. Here's an example:

```
somefont = createFont('font_name.ttf', 20)
```

Be sure to include the file extension.

To activate the new font, use `textFont()`. Then, draw the pangram once more (the third version) to confirm that it's working:

```
textFont(seriffont)
text(pangram, 0, 150)
```

The `textFont()` function accepts a single argument, a Processing-readied font. All subsequent text() functions will use the `seriffont` until Processing encounters another `textFont()` function.

The `textLeading()` function controls the leading of your text. *Leading* (which rhymes with *wedding*) is the typographic term to describe the spacing between each line of text.

The `textAlign()` function controls text alignment; you can use an argument of `LEFT`, `CENTER`, or `RIGHT` to set the horizontal alignment of your text.

You'll use the `textLeading()` and `textAlign()` functions to render the bottom two (fourth and fifth) versions of the pangram in Figure 3-6. Add a left- and right-aligned pangram:

```
❶ textLeading(10)
  text(pangram, 0, 200, 250, 100)
  textAlign(RIGHT)
  text(pangram, 0, 250, 250, 100)
```

The first pangram is left-aligned because that's the Processing default. I've added width and height arguments to the `text()` functions to invoke word wrapping. Each pangram is constrained to its own rectangular area that's 250 pixels wide by 100 pixels high. If a line of text exceeds the width of 250 pixels, Processing automatically pushes the words that don't fit onto a new line. If any lines even partially exceed the height of the text area (100 pixels), you do not see them, although this doesn't happen here. The leading is reduced to 10 pixels ❶, causing the lines to overlap. Ordinarily, the leading value is proportionate to the font size.

Just like fill, stroke, and many other Processing attributes, the text parameters you set remain in effect until you specify otherwise. But if you adjust the text size—using another `textSize()` function—the leading will reset to a proportional value.

Summary

This brief introduction explored manipulating strings by using Python's slice notation and string methods, and drawing text in the display window with Processing's `text()` function. Processing's typography functions allow you to control font size, horizontal alignment, line spacing/leading, and font selection. You'll be using string methods and text functions in many of the tasks to come.

In Chapter 4, you'll explore topics including control flow and conditional statements—techniques that allow you to write programs that can skip, jump to, and repeat lines of code. These tools are helpful because they let you change the order of your code's execution, and whether it executes at all, based on specific rules and values.

4

CONDITIONAL STATEMENTS
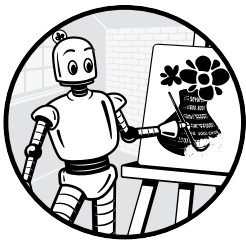
The programs you have written so far execute line by line, beginning at the top of the code and ending at the bottom. You can visualize this flow as a series of steps that execute in a linear fashion, which means that the program can run in only one way. In this chapter, you'll explore how to write divergent paths for Python to follow, depending on whether certain conditions are met. This is useful because you can execute different actions in your program depending on the scenario—think of the way a video game directs you to different levels or screens contingent on your performance.

To evaluate a condition, you'll use the *Boolean* data type, which represents one of two states: true or false. You'll learn to write Boolean expressions to test whether a statement is true or false. Then you'll use if, elif, and else statements to make your code carry out different actions in response to the true or false outcomes.

Control Flow

Control flow refers to the order in which your lines of code execute. By default, this flow begins from the top of your code and proceeds one line at a time until it reaches the bottom. Using control flow statements like `if`, `elif`, `else`, `while`, and `for`, you can direct Python to skip, jump to, and repeat lines of code.

For example, say you want to fill the display window with circles. Figure 4-1 depicts two arrangements: 9 circles aligned three by three, and 81 circles aligned nine by nine.
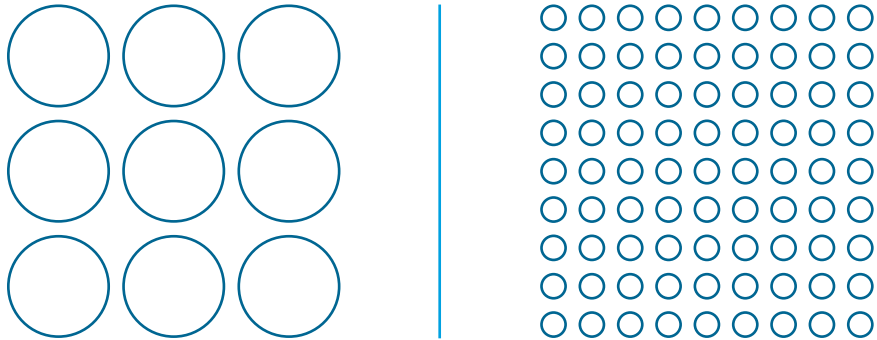
Figure 4-1: The 9-circle (left) and 81-circle (right) arrangements

You could write a `circle()` function for each and every circle displayed. If you're drawing only 9 circles, writing 9 `circle()` functions might be manageable, but writing 81 `circle()` functions is tedious and can lead to errors. If you want several circles, the better approach is to write a single `circle()` line and have Python repeat it as many times as needed. Figure 4-2 shows these two approaches, using flowcharts representing the programming logic.
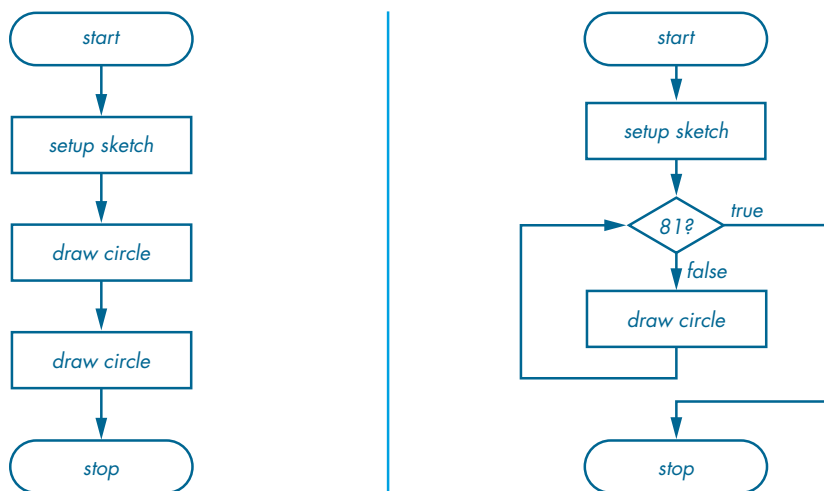

Figure 4-2: Flowcharts comparing manual (left) and conditional (right) approaches to drawing multiple circles

The manual method is shown on the left. Each *draw circle* represents a circle() function; in this case, there are two *draw circle* steps, but you can add as many as required.

The flowchart on the right in Figure 4-2 repeats the *draw circle* step until a particular condition is met. The diamond containing *81?* represents a decision step, which checks whether the current number of circles is 81. If true, the program proceeds to the *stop* step; if false, Processing draws another circle and returns to the decision step.

This chapter and the next examine how to implement this kind of logic in Python, which will be your first foray into algorithmic thinking. In later chapters, you'll be applying flow control techniques in most of your sketches.

Conditional Statements

Conditional statements are used to test one or many conditions and then execute appropriate responses.

To explore Python's various conditional statements, create a new sketch and save it as *conditional_statements*. In the sections that follow, you'll enter code into this working sketch.

The Boolean Data Type

As mentioned previously, a Boolean is a value that can represent one of two possible states: True or False. To see how the Boolean data type operates, add these two variables to the sketch:

```
ball_is_red = True
ball_is_spiky = False
```

The first letter of a Boolean value is always uppercase, and no quotation marks are used since that would make it a string.

Whenever Python is required to manage Booleans as numeric values, it converts a True value to 1, and a False value to 0; this, however, works both ways. For instance, Python's bool() function, which converts any value to Boolean, converts a 1 to True and a 0 to False. This will prove useful when you encounter if statements, where you'll instruct Python to execute different lines of code based on True/False outcomes.

In your sketch, add a series of print() functions to test this behavior:

```
. . .
print(ball_is_red)        # displays: True
print(ball_is_spiky)      # displays: False
print(ball_is_red + True) # displays: 2
print(bool(1))            # displays: True
print(bool(0))            # displays: False
```

The first two print statements repeat the variable values back to the console. The third print statement uses an arithmetic addition (+) operator

to add one `True` Boolean to another. Adding `True` and `True` results in `2`. Converting Booleans to numbers works with mathematical operators or any functions that convert values to numbers, such as the `int()` function for converting to integers. The final two `print` statements, which contain `bool()` functions, convert `1` and `0` to their respective Boolean equivalents.

Relational Operators

The previous example explicitly defines whether the ball is red and/or spiky, but relational operators also can direct your program to make its own decisions as to what is true or false. *Relational operators*, like greater-than (>) and less-than (<) signs, determine the relationship between two operands. For example, given 3 > 2, the 3 and 2 are the operands, and the greater-than sign is the relational operator. Because 3 is indeed greater than 2, this statement is true.

To see how this works, add the following code to your *conditional_statements* sketch:

```
. . .
x = 2
print(x > 1)  # displays: True
print(x < 1)  # displays: False
```

The variable x is equal to `2`, which is greater than `1`, so the console should display `True`. However, `2` is not less than `1`, so the final line should print `False`. Notice that the relational operators return a Boolean value. This will be important for the next section, where the results of such comparisons determine which lines of code your program will execute. Table 4-1 shows a list of Python's relational operators.

Table 4-1: Relational Operators

Operator	Description	Example
>	Left operand is greater than right	2 > 1 returns True
<	Left operand is less than right	1 < 2 returns True
>=	Left operand is greater than or equal to right	1 >= 2 returns False
<=	Left operand is less than or equal to right	2 <= 2 returns True
==	Left operand is equal to right	2 == 2 returns True
!=	Left operand is not equal to right	2 != 2 returns False

What Table 4-1 doesn't show is that the `==` and `!=` operators can operate on both numbers and strings. Add the following code to test this:

```
. . .
name = 'Jo'
print(name == 'Jo')  # displays: True
print(name != 'Em')  # displays: True
```

Next you'll combine relational operators with `if` and other conditional statements to specify the conditions for executing lines of code.

if Statements

The `if` statement requires two ingredients: an expression that returns `True` or `False`, and code to execute should the former evaluate as `True`. Figure 4-3 illustrates the syntax of an `if` statement.
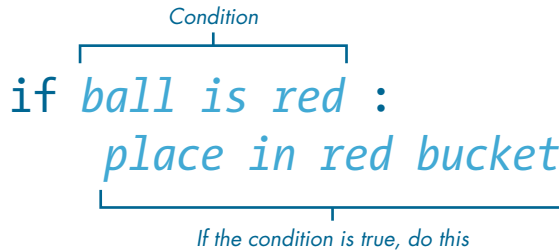

Figure 4-3: An `if` statement syntax

Everything in pale blue is placeholder *pseudocode*, which is just English text that describes what's happening in the code; the idea is that you could later replace this with Python.

Assigning a Passing Grade

To get started with `if` statements, you'll build a simple program that assigns letter grades to students, depending on their percentage test scores. Begin by adding this code to your working sketch:

```
. . .
score = 60

if score >= 50:
    print('PASS')
```

This awards a `PASS` grade for any score greater than or equal to 50. In this instance, the `score >= 50` returns `True`, so the `print('PASS')` line is executed. Be sure to indent the `print` line, which you can do by using the TAB key.

NOTE   *Whenever you press TAB, the Processing editor inserts four spaces and not a tab character. Python permits indentation using any number of space or tab characters, provided that you're consistent, but two or four spaces is most common. That said, the Processing editor will automatically indent using four spaces as you press ENTER after any line that ends in a colon (:). I recommend sticking to Processing's default.*

Everything indented beneath the `if` line is executed if the condition returns `True`. For example, add the following line to your code:

```
. . .
if score >= 50:
    print('PASS')
    print('Well done!')
```

This should now print both `PASS` and `Well done!` for any score greater than or equal to 50.

On the other hand, a `print` line flush against the left margin prints `Well done!`, regardless of whether the score exceeds 50:

```
. . .
score = 10

if score >= 50:
    print('PASS')
print('Well done!')
```

If you ever need to nest an `if` statement within another `if` statement, increase the indentation accordingly. Most code editors allow you to select multiple lines of code and press the TAB key to indent them simultaneously. Processing's editor is no exception. If you need to "out-dent," hold the SHIFT key while pressing TAB.

Without adding this next example to your code, see if you can predict the result:

```
  score = 60
❶ language = 'ES'  # for Español (Spanish)

❷ if score >= 50:
    ❸ print('PASS')

    ❹ if language == 'EN':
          print('Well done!')

    ❺ if language == 'ES':
          print('Bien hecho!')
```

If you predicted that the console would display a `PASS` line followed by `Bien hecho!`, you're correct. The `'ES'` string value is assigned to a new variable named `language` ❶. The score is greater than or equal to 50, so the program executes the contents of the outermost `if` statement ❷. The `PASS` line ❸ is first to print. The condition of the next `if` statement ❹, however, evaluates as `False`, so the program skips the `Well done!` line. The final `if` statement ❺ then tests for Spanish. Because the language variable is equal to `'ES'`, Processing prints `Bien hecho!` to the console.

Assigning Letter Grades

Currently, your grading program can award only a PASS. To assign letter grades like A, B, or C, you'll need to use additional `if` statements.
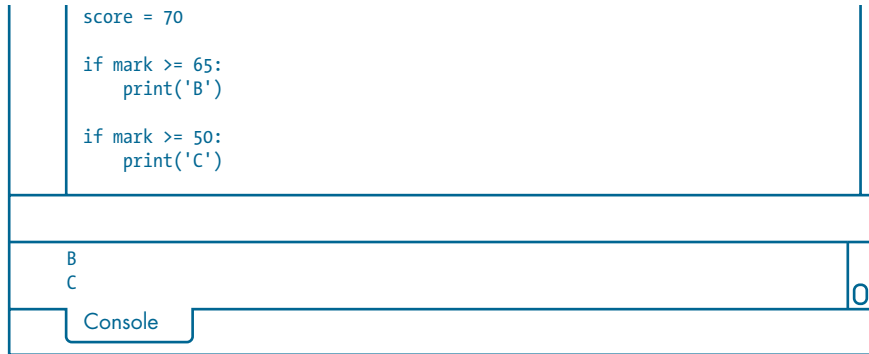
Adapt your code, changing the PASS string to a C and inserting a new `if` statement that awards a B for any score greater than or equal to 65:

```
. . .
score = 60

if score >= 65:
    print('B')
```

```
if score >= 50:
    print('C')
```

Run the sketch. Because the score variable holds a value greater than 50, the console displays a C. But there's an issue—when you change the score value to a 70, you get B *and* C (Figure 4-4).

```
score = 70

if mark >= 65:
    print('B')

if mark >= 50:
    print('C')
```

```
B
C
```
Console   0

Figure 4-4: A score of 70 is awarded both B and C.

Because the score is greater than 65 and greater than 50, both if statements are triggered to print, resulting in two letter grades. To avoid getting more than one grade, you need to chain together the if statements, such that, if the first condition is found to be true, the subsequent if statement is skipped. This is where the else-if style structure comes into play.

elif Statements

An *else-if statement*, or elif in Python, runs only after an if condition returns False. Using an elif will solve the preceding problem of having both if statements operate independently. So, just change the second if to an elif:

```
. . .
score = 60

if score >= 65:
    print('B')

elif score >= 50:
    print('C')
```

Now, if the value of score results in B, there is no need to check the C condition, and Python will skip the elif statement altogether. On the other hand, should the initial if statement condition return False, the elif will test whether score is greater than or equal to 50; if so, it prints a C.

Set the score variable to something B-worthy, like 70, and then run the sketch. The console should now display a B, but no C.

Order Matters

It's important to order this `if...elif` logic correctly. Consider, for instance, the following code that places the `C` condition first:

```
score = 70

if score >= 50:
    print('C')

elif score >= 65:
    print('B')
```

In this scenario, any score greater than or equal to 50 gets a C, even if it's higher than 65 and should get a B. In fact, no B grades will ever print to the console, because the program can never check the `B` condition.

Checking for A

In your *conditional_statements* sketch, insert a new `if` statement to handle A grades (80 or greater). Also, change the `B` statement to an `elif`. Adjust the `score` to test that this is working correctly:

```
score = 87

if score >= 80:
    print('A')

elif score >= 65:
    print('B')

elif score >= 50:
    print('C')
```

You can add as many `elif` statements as you need, but it's always a single `if` that marks the beginning of the `if...elif` chain.

Your A/B/C logic is now in place, but a score below 50 will pass through all of the `if...elif` statements without invoking any actions, not receiving any grade at all.

else Statements

If a student does not receive an A, B, or C, you can conclude that the grade is a `FAIL`. To handle `FAIL` cases, use an *else statement* to account for any condition that doesn't match those in the `if...elif` grouping. You don't need to check whether the `score` variable is less than 50, as this is implied by its failing to match any of the preceding criteria. To handle `FAIL` cases, add the following `else` statement to your code:

```
. . .
    print('C')
```

```
else:
    print('FAIL')
```

The else statement has no condition and always comes at the end of the
if...elif grouping.

Adjust the score value to something like 40 and test the code. The con-
sole should display a FAIL.

else Statements Without elif

An else statement need not necessarily follow an elif. You can use an if...
else structure where you don't require elif clauses. Consider a program
that grades any score 50 and above as a PASS, and everything else as a FAIL:

```
if score >= 50:
    print('PASS')
else:
    print('FAIL')
```

There are no elif statements, so the else handles any score that the
if doesn't catch. Of course, whether you include any elif statements all
depends on the logic you intend to implement.

Logical Operators

So far, each if...elif statement has relied on the outcome of a single
relational operation. But often it's useful to evaluate multiple relational
operations within a single expression. For example, you might want to
check whether the ball from earlier is red *and* spiky. To do this, you can
use *logical operators*, or an if statement nested within another if statement,
to make a decision based on the outcomes of multiple conditions.

Let's modify the earlier example to handle red, spiky balls. You could
use a nested if statement as shown in the following pseudocode:

```
if ball is red:

    if ball is spiky:
        place in red & spiky bucket
```

The outer if checks whether the ball is red, and then the inner if
checks whether it's spiky.

You can do the same thing by using a single if statement with the
logical operator and:

```
if ball is red and ball is spiky:
        place in red & spiky bucket
```

The and operator returns True if the expressions on both sides of the
operator also return True.

Table 4-2 provides a list of Python's logical operators, along with a brief description of each and an example.

Table 4-2: Logical Operators

Operator	Description	Example
and	Returns True if both operands are true	`2 > 1 and 4 > 3` returns True
or	Returns True if at least one operand is true	`2 > 1 or 4 < 3` returns True
not	True becomes False, and vice versa	`not 4 < 3` returns True

Now let's join two expressions by using the `and` operator to check for a much narrower condition. Add another `if` statement to check whether the student's score is greater than or equal to 45 *and* less than 50, and display OFFER RETAKE if so:

```
. . .
if score >= 45 and score < 50:
    print('OFFER RETAKE')
```

This condition includes the logical operator `and`. For this to evaluate as True, both `score >= 45` and `score < 50` must return True. Change the `score` value to something within this range, like `46`, and confirm that the console displays FAIL and OFFER RETAKE.

Checking for Invalid Input

Now, let's add another `if` statement that uses the `or` condition to account for any scores outside the valid range (0 to 100). Place this at the top of the `if...elif` chain, and change the A statement to an `elif`:

```
. . .
score = 105

❶ if score < 0 or score > 100:
    print('INVALID SCORE')

❷ elif score >= 80 :
    print('A')
. . .
```

Now, if the `score` is `105`, the program should print INVALID SCORE. For the `or` operation ❶ to evaluate as True, at least one of its two operands, `score < 0` and `score > 100`, must return True. Experiment with different score values to test the code. If the console displays the invalid score message as well as a grade, ensure that you've changed the second statement to an `elif` ❷.

Displaying a Message for Invalid Input

There is room for one last improvement. Currently, if the user enters a score of 0, the program grades it as a FAIL. However, a score of 0 is relatively unusual, so add one final if statement to display a warning message that the user may have entered invalid input.

Here is the complete grading program:

```
. . .
score = 0

if score < 0 or score > 100:
    print('INVALID SCORE')
elif score >= 80:
    print('A')
elif score >= 65:
    print('B')
elif score >= 50:
    print('C')
else:
    print('FAIL')

if score >= 45 and score < 50:
    print('OFFER RETAKE')

if not❶ score:
    print('WARNING: SCORE IS ZERO')
```

Recall that, when dealing with Booleans, Python interprets a 0 as False. This means if the score is assigned a 0, it's evaluated as False. However, the not operator ❶ reverses this Boolean, converting it to a True, thereby triggering the warning message. You could use mark == 0 to test the same condition, which is more explicit and easier to read, but this was a good opportunity to show the not operator in action. Note that this is a separate if statement, so for a score of 0, the console displays both the FAIL and a warning message.

You might decide that the program could use some clickable buttons and input fields. Chapter 11 covers mouse and keyboard interaction techniques you can use to add a graphical user interface to this kind of program.

Challenge #3: Four-Square Task

In this challenge, you'll use conditional statements to gauge the position of a point in a four-colored square. Add the following code:

```
size(600, 600)
noFill()
noStroke()

fill('#FF0000')  # red quadrant
rect(width/2, 0, width/2, height/2)
```

```
fill('#004477')  # blue quadrant
rect(0, 0, width/2, height/2)

fill('#6633FF')  # violet quadrant
rect(0, height/2, width/2, height/2)

fill('#FF9900')  # orange quadrant
rect(width/2, height/2, width/2, height/2)
```

Run the sketch. The display window should appear, equally divided into different-colored quadrants (Figure 4-5). Each fill() and rect() line pair draws a colored square spanning from the center of the display window to each corner.

Figure 4-5: Grid with four colors

Next, place a single text character in the upper right quadrant:

```
. . .
x = 400
y = 100
❶ txt = '?'

fill('#FFFFFF')
textSize(40)
textAlign(CENTER, CENTER)
❷ text(txt, x, y)
```

The txt value ❶, a question mark, is positioned in the red quadrant (Figure 4-6). The text() function ❷ relies on the x and y variables to control this character's position.
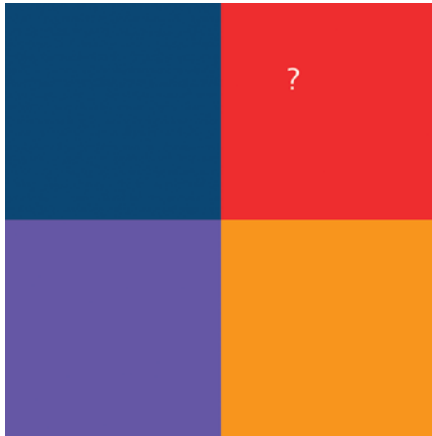
Figure 4-6: Placing the question mark in the red (upper right) quadrant

Your challenge is to write conditional statements to replace the ? character with an R (for red), B (for blue), P (for purple), or O (for orange) in the appropriate squares to match the color beneath it.

To start, insert an if statement to manage the R (red) condition:

```
. . .
txt = '?'

if x >= width/2:
    txt = 'R'

fill('#FFFFFF')
. . .
```

The if statement sets the txt variable to 'R' for any x values to the right of the display window's center. Run the sketch to confirm that the code displays an R over the red quadrant. If you're still seeing a question mark, ensure that you have inserted the if statement above the fill('#FFFFFF') line.

Now, set the y value to 400 to place the character in the orange quadrant. Run the sketch. It's still an R. To display an O instead, you need to add an elif statement (and a logical and operator). Once the O is working correctly, try positioning the character in another quadrant, and so forth. Figure 4-7 displays four screenshots of the completed task; the caption lists the corresponding x-y coordinates.

Now that you have a good grasp of if...elif...else logic, you're ready to use Boolean expressions for iteration. If you need help, you can access the solution at *https://github.com/tabreturn/processing.py-book/tree/master/ chapter-04-conditional_statements/four_square/*.
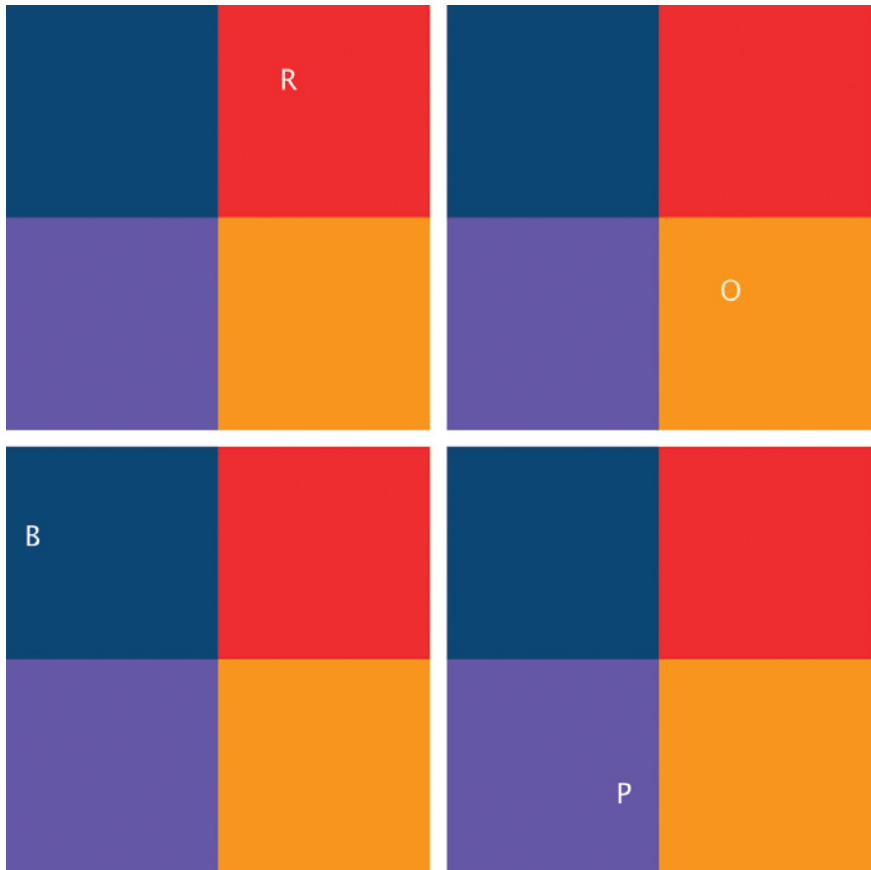
Figure 4-7: Clockwise from the upper left: the x-y coordinates of each letter are
R = (400, 100), O = (400, 400), P = (250, 485), and B = (38, 121).

Summary

In this chapter, you learned about the Boolean data type, relational opera-
tors, and how to write Boolean expressions that work with `if` statements to
instruct Python to execute particular lines of code. You explored logical
operators for constructing richer expressions, as well as how to combine `if`,
`elif`, and `else` statements.

In Chapter 5, you'll take control flow a step further and learn how to
write programs that can repeat an operation until a certain requirement is
met. For some especially interesting results, you'll be adding randomness
to your creations.

5

ITERATION AND RANDOMNESS
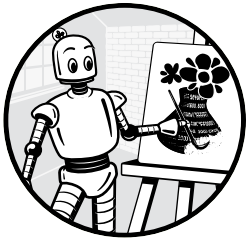
In Chapter 4, you learned how to program divergent paths for Python to follow. In this chapter, you'll create looped paths with `while` and `for` loop statements. *Loop statements* repeat actions, so you don't need to rewrite the same or similar code multiple times, resulting in fewer lines of code. In other words, you can solve problems more efficiently with code that's easier to adapt. You'll use these loop statements to generate visual patterns in Processing.

You'll also learn to apply randomness to your patterns to make them more compelling and unpredictable. Processing's `random()` function is useful for generating randomized arguments in your shape functions, allowing you to create irregular designs. You can also randomize the conditions for your control flow statements so that your code executes differently on each run. Randomness is, undoubtedly, one of the most useful and exciting tools in the creative coder's toolset, because it allows you to write programs that can produce unpredictable results.

Iteration

In computer programming, *iteration* is the process of repeating a series of instructions a specified number of times or until a condition is met. As an example, say you want to tile a floor. Starting in one corner, you lay one tile. Then you place another tile next to it, repeating the process until you've reached the opposite wall, at which point you move down a row and continue. In this scenario, placing an individual tile is a single iteration. In many iterative processes, the result of a previous iteration defines the starting point of the next.

Tasks like tiling can be tedious work, though. Humans are exemplary in reasoning and creative thought, but if not sufficiently stimulated, they tend to lose interest in performing such monotonous activities. Computers, however, excel at performing repetitive tasks rapidly and accurately, especially when numbers are involved.

Using Iteration to Draw Concentric Circles

To begin exploring iteration in Processing, create a new sketch and save it as *concentric_circles*. Add the following code:

```
size(500, 500)
background('#004477')
noFill()
stroke('#FFFFFF')
strokeWeight(3)

circle(width/2, height/2, 30)
circle(width/2, height/2, 60)
circle(width/2, height/2, 90)
```

Each `circle()` function has its x-y coordinate placed in the center of the display window. The first circle is the smallest, with a diameter argument of 30; each subsequent circle is 30 pixels larger in diameter than the one preceding it. The program runs each `circle()` function line by line, advancing toward a display window filled with concentric circles (Figure 5-1).
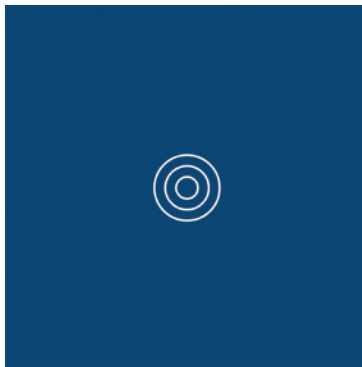
Figure 5-1: Three circles rendered using three `circle()` functions

However, to fill the entire window, you'd need to write many more circle() lines. Instead of adding circle() functions manually, you can use a Python while loop to run them iteratively.

while Loops

A while loop is a control flow statement that looks and behaves much like if. The key difference is that while continues to execute the lines indented beneath it *until* its accompanying condition is no longer true.

Back in your *concentric_circles* sketch, comment out the circle() lines by using ''' for multiline comments, and add a basic while loop structure:
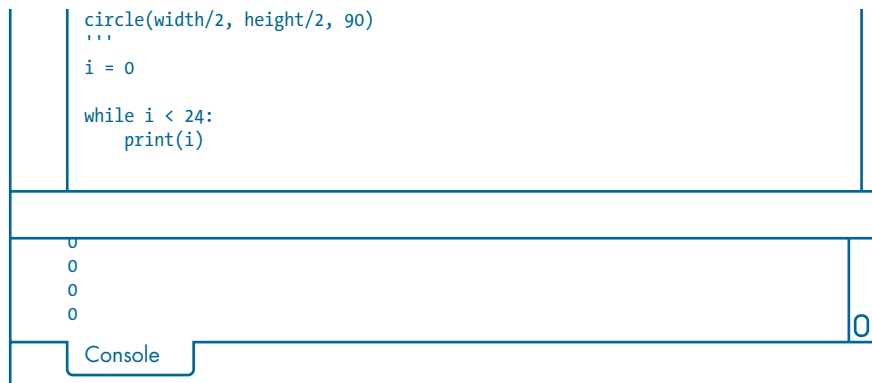
```
. . .
'''
circle(width/2, height/2, 30, 30)
circle(width/2, height/2, 60, 60)
circle(width/2, height/2, 90, 90)
'''
i = 0

while i < 24:
    print(i)
```

The i variable is defined to serve as your *loop counter*, controlling the iterations of the while statement. For the while expression, i is equal to 0 and, therefore, is less than 24. Unlike an if statement that would execute the print() function a single time, the while repeatedly executes the print line until the value of i reaches 24—which, in this case, is never.

NOTE   *As with any other variable, you can name i whatever you like, but it's a popular convention to represent a loop counter value with an i.*

Running the sketch should print an endless list of 0 digits to the console (Figure 5-2).

Figure 5-2: The console lists endless lines of zeros.

This code has crashed your program by sending it into an infinitive loop! To exit the program, click the **Stop** button. Processing may take some time to respond. The variable i remains 0, and the i < 24 condition never achieves the False required to conclude the loop.

To correct this, add 1 to i with each iteration of the while loop:

```
. . .
while i < 24:
    print(i)
    i = i + 1
```

This new line states that the loop counter, i, is equal to itself plus 1. On the first iteration, i is 0, which is less than 24, so the program prints 0, adds 1 to i, and then begins the process again. On the next iteration, i is 1, which is still less than 24, so the program prints 1, adds 1 to it, and restarts the process. The iteration continues as long as i < 24 evaluates to True. Once i reaches 24, the program exits the loop and runs any other code that follows the while block.

Note that the output never reaches 24 (Figure 5-3), because the while condition states "where i is less than 24," not "less than or equal to 24."

To draw 24 circles, place a circle() function within the loop:

```
. . .
while i < 24:
    print(i)
    circle(width/2, height/2, 30*i)
    i = i + 1
```
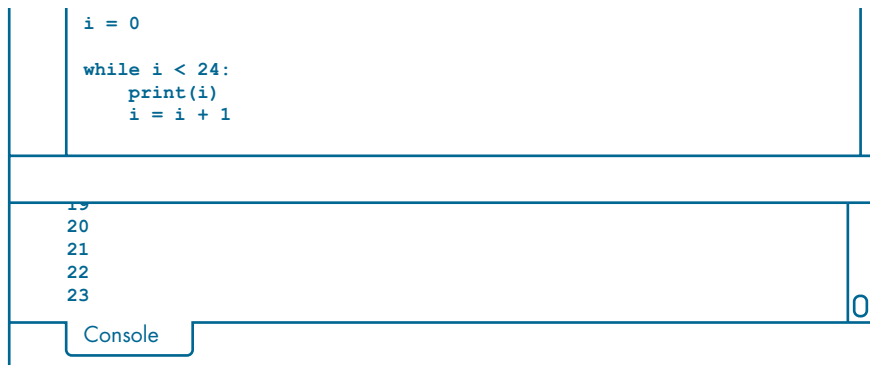

Figure 5-3: The console displays 0 to 23, but not 24.

To avoid drawing 24 circles of exactly the same size, in the same position, use i as a multiplier for the circle() diameter argument. On the first iteration, the diameter argument is equal to 30*0. Therefore, the first circle, placed in the very center of the display window, has a diameter of 0 and doesn't render (Figure 5-4).

The other 23 circles are enough to fill the 500 × 500 pixel area. By changing the number in the while statement, you may draw as many (or as few) circles as you like.
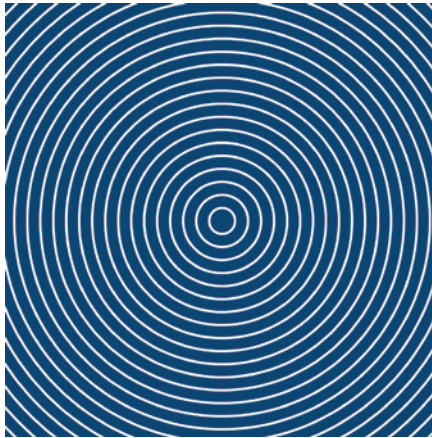
*Figure 5-4: The drawing now has 24 circles
(one invisible, and some partially cropped).*

AUGMENTED ASSIGNMENT OPERATORS

You're already familiar with the = operator (assignment), but not its arithmetic variants. In the *concentric_circles* example, you incremented i by using this line of code:

```
i = i + 1
```

This states that i is equal to itself plus 1. To simplify this statement, you can instead write this:

```
i += 1
```

The result is exactly the same, but the latter is easier to read and write. Table 5-1 provides a list of these *augmented assignment operators*, along with an example of each.

Table 5-1: Augmented Assignment Operators

Operator	Example
+=	i += 1 is equivalent to i = i + 1
-=	i -= 1 is equivalent to i = i - 1
*=	i *= 1 is equivalent to i = i * 1
/=	i /= 1 is equivalent to i = i / 1

for Loops

The Python `for` loop executes a given block of code a specified number of times. Unlike the `while` loop that relies on a conditional expression, the `for` loop iterates a sequence. A *sequence* is a collection of values; for instance, string data is a sequence of characters. Python *lists* are particularly versatile sequences, which I cover in Chapter 7. To generate sequences for the `for` loops in this section, you'll use the `range()` function.

A `for` loop is more appropriate than a `while` loop when you've established the number of iterations required before entering the loop. Generally speaking, the `for` loop is shorter and simpler, and won't trigger infinite loops. When either a `while` or `for` will do, opt for the `for` loop.

One of the easiest ways to understand the `for` loop is to convert something you already wrote that uses a `while` statement. Save *concentric_circles* as a new sketch called *for_loop* by using **File ▸ Save As**. Comment out the `while` loop parts, and add the following for loop:

```
. . .
'''
❶ i = 0

  while i < 24:
      print(i)
      circle(width/2, height/2, 30*i)
❷     i = i + 1
  '''
❸ for i in range(24):
      print(i)
      circle(width/2, height/2, 30*i)
```

In the `while` loop version, recall that you had to define the `i` variable to serve as a loop counter. With each iteration of the `while` block, you also had to increment `i` to avoid entering an endless loop. The `for` statement does away with the need to define and manage a separate counter variable.

So, `i = 0` ❶ is no longer necessary, nor is the nested statement to increment it ❷. Instead, the `range()` function takes its argument of 24 to generate a sequence from 0 up to but not including 24 that controls the `for` ❸ loop iteration behavior. On the first iteration, `i` is equal to 0, the first value in the sequence. With each subsequent iteration, the next value in the `range()` sequence is assigned to `i`. When `i` reaches 23, the `for` block runs for the last time, and then Python exits the loop. Run the sketch to confirm that the display window looks the same as Figure 5-4.

The `range()` function can handle up to three parameters. Provide two arguments for a start and end value, respectively:

```
. . .
for i in range(10, 13):
    print(i)
    circle(width/2, height/2, 30*i)
```

In this instance, the `circle()` function should execute three times, for i = 10, i = 11, and i = 12. Run the sketch to see the result (Figure 5-5). You should see three concentric rings.


```
for i in range(10, 13):
    print(i)
    circle(width/2, height/2, 30*i)
```
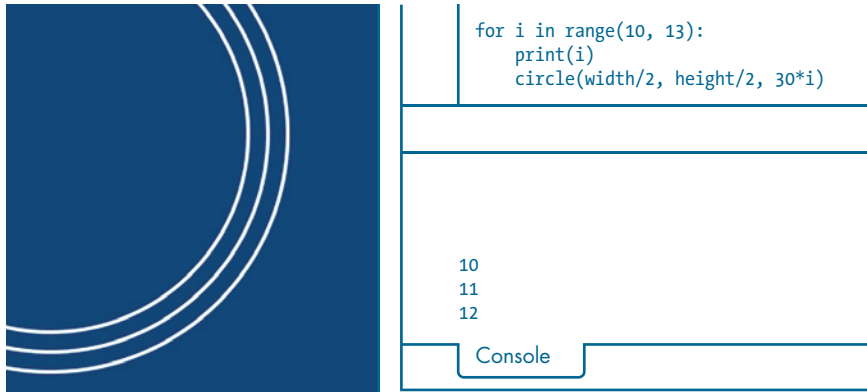
```
10
11
12
```

Console

Figure 5-5: Result for `range(10, 13)`

Now use three range arguments to represent a start, end, and step size, respectively. The *step size* is the difference between each integer in the sequence:

```
. . .
for i in range(3, 13, 3):
    print(i)
    circle(width/2, height/2, 30*i)
```

In this instance, the `circle()` function should execute four times, for i = 3, i = 6, i = 9, and i = 12. The result should be four rings with enlarged spacing (Figure 5-6).


```
for i in range(3, 13, 3):
    print(i)
    circle(width/2, height/2, 30*i)
```

```
3
6
9
12
```

Console
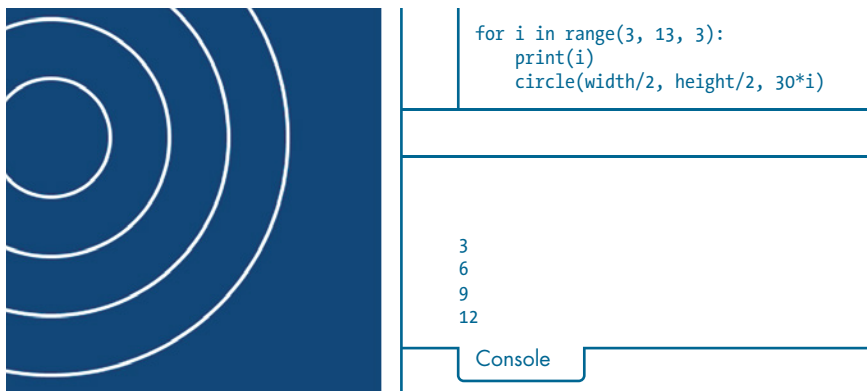
Figure 5-6: Result for `range(3, 13, 3)`

Experiment with different range arguments to see how the circles are affected.

Challenge #4: Create Line Patterns

In this challenge, recreate the three patterns shown in Figure 5-7 by using the line() function and one for loop for each. Don't worry if your code produces a slightly different result, as long as the basic pattern remains the same.

If you're not sure where to begin, here are a few clues to help you approach each pattern:

- The left pattern is similar to the concentric circles, except it has 12 diagonal lines.
- For the middle pattern, the line spacing increases by a multiple of 1.5 with each for loop iteration. Defining an additional variable may help.
- The right pattern requires an if...else structure nested within the for loop. You might consider using a modulo (%) operator, described in Chapter 1, to establish whether i is odd or even.

If you need help, you can find the solution at *https://github.com/ tabreturn/processing.py-book/tree/master/chapter-05-iteration_and_randomness/ for_loop_patterns/*.
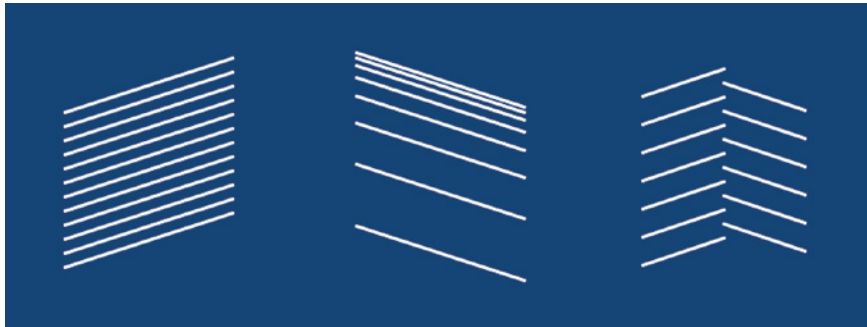
Figure 5-7: Three for loop patterns

break and continue Statements

Loops provide an efficient way to automate and repeat tasks. Sometimes, though, you need to exit a loop prematurely. For example, when you draw a series of concentric circles to fill the display window, like in the earlier task, you might want to *break* the loop if the circles reach the edge of the display window before exhausting the sequence of range() values. If Python encounters a break statement within a for or while loop, it will immediately terminate the loop. Once the loop is terminated, your program will move along as usual.

Sometimes you need to terminate an iteration (not the entire loop), prompting Python to begin the next iteration immediately. For this, use the continue statement.

Let's look at a brief example comparing an ordinary loop, a loop with a break statement, and a loop with a continue statement. There's no need to write any code. Figure 5-8 depicts three dotted lines, drawn from left to right using each type of loop.

The loop for the pale blue (top) dotted line looks like this:

```
for i in range(20, width, 20):
    fill('#0099FF')
    circle(i, 75, 10)
```

With each iteration, the circle() function draws a new dot, placing it 20 pixels to the right of its predecessor. The first dot has an x-coordinate of 20; the loop completes as the dotted line reaches the width of the display window. This loop is not concerned with the two vertical red bands and draws dots right through them.
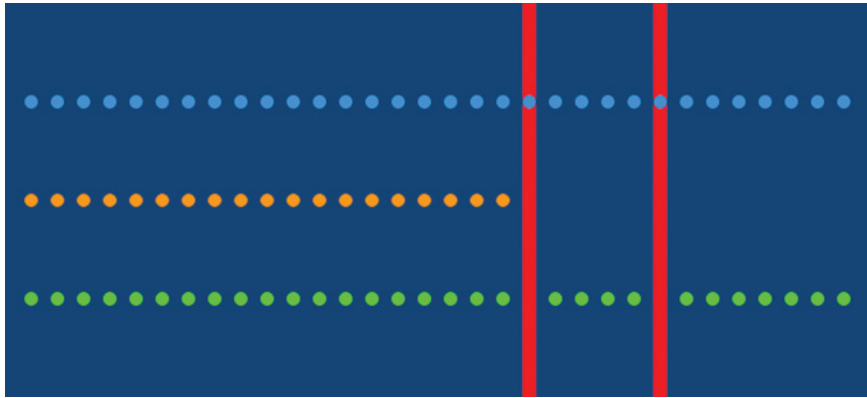
Figure 5-8: Drawing dotted lines using different loops

The loop for the orange (middle) dotted line looks like this:

```
for i in range(20, width, 20):

    if red(get(i, 150)) == 255:
        break

    fill('#FF9900')
    circle(i, 150, 10)
```

The get() function accepts an x-y coordinate and returns the color for the pixel at that position; to extract the red value for the pixel, you wrap the get() function with a red() function. This will return a red value between 0 and 255 based on the RGB mixture, which means a value of 255 for any pixels in the bright red bands (#FF0000). The loop will check for a red pixel before it draws a dot; if detected, the break statement will terminate the loop. The fill() and circle() functions do not draw a dot on the final iteration, because the break statement exits the loop immediately.

The loop for the green (bottom) dotted line looks like this:

```
for i in range(20, width, 20):

    if red(get(i, 225)) == 255:
        continue

    fill('#00FF00')
    circle(i, 225, 10)
```

This loop will check for a red pixel before it draws a dot; if the pixel is detected, the continue statement immediately terminates the current iteration of the loop to start at the beginning of the next, skipping over the fill() and circle() functions.

Randomness

Randomness is an important concept in computer programming because of its applications in cryptography. Moreover, randomness is programmed into everything from video games to simulations to gambling software. However, computer-generated random numbers aren't truly random, because they're created using a specific algorithm. If you know the algorithm and the conditions used to generate "random" numbers, you can predict patterns in the sequence. Therefore, a computer can simulate randomness only by generating *pseudorandom* numbers, which are not truly random but statistically similar enough to actual random numbers.

In this section, you'll use the Processing random() and randomSeed() functions to generate pseudorandom values. With these randomized values, you'll draw more interesting patterns than you might be able to create with predefined values.

random() Function

Each time you call Processing's random() function, it produces an unexpected value within a specified range. To begin experimenting with randomness, create a new sketch and save it as *random_functions*. Add the following setup code:

```
size(600, 250)
background('#004477')
noFill()
stroke('#FFFFFF')
strokeWeight(9)
```

The new sketch has a blue background. Soon, you'll draw points; the size of your points is affected by the strokeWeight() function.

The `random()` function can accept up to two arguments. In the case of a single argument, you're defining an upper limit:

```
print(random(5))
```

This code will display a random floating-point value ranging from 0 up to but not including 5.

Two arguments represent an upper and lower limit, respectively:

```
print(random(5, 10))
```

This time around, the console displays a random floating-point number ranging from 5 up to but not including 10.

If you want a random integer instead, wrap the `random()` function in `int()`. This converts the floating point to an integer by removing the decimal point and everything that follows it:

```
print(int(random(5, 10)))
```

Figure 5-9 shows what you can expect to see. Of course, given that the values are random, the console output will appear differently on your computer, as well as each time you run the sketch.

```
print(random(5))
print(random(5, 10))
print(int(random(5, 10)))
```

                                                                    random_fu

```
2.21987342834
8.80465507507
6
```
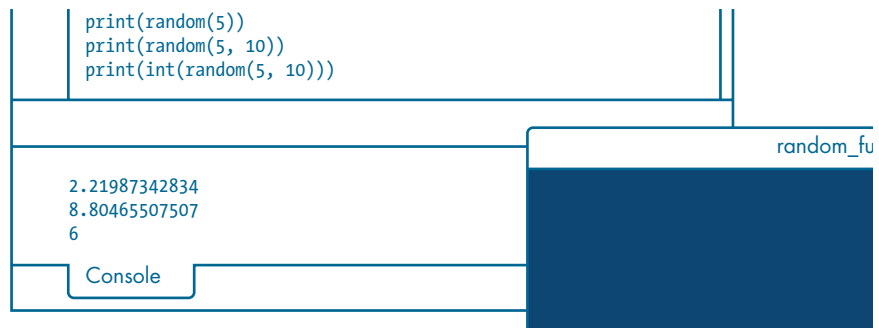
Console

Figure 5-9: Experimenting with different random() arguments

Next, let's generate 50 random values. Rather than print a long list in the console area, plot them as a series of points sharing a y-coordinate. Add the following code:

```
. . .
for i in range(50):
    point(random(width), height/2)
```

This `point()` function uses the `random()` function to define its x-coordinate. The y-coordinate is always `height/2`. The points should distribute differently each time you run the sketch (Figure 5-10).

Figure 5-10: Random values distributed along a line

Now change the `range` argument from 50 to 500, and plot the point using random x- and y-coordinates:

```
. . .
for i in range(500):
    point(random(width), random(height))
```

The result should be a display window filled with 500 randomly positioned points (Figure 5-11).
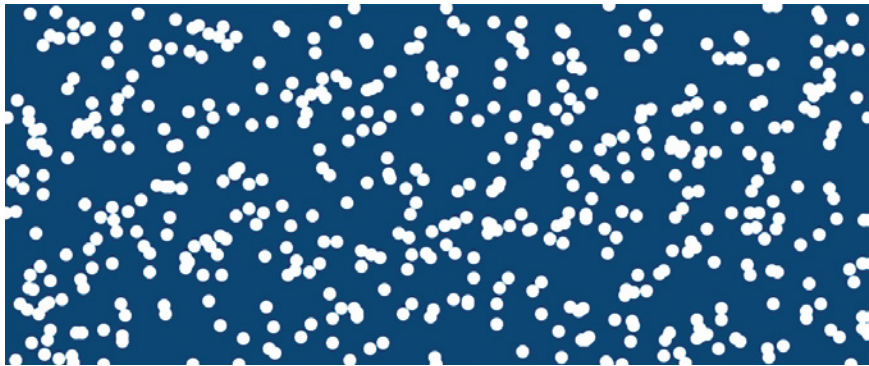
Figure 5-11: Filling the display window with randomly positioned points

Each time you run the sketch, it produces a (slightly) different arrangement.

Random Seed

In Figures 5-10 and 5-11, Processing picks the coordinates from a pseudo-random *sequence* of numbers. This pseudorandom sequence itself relies on a *random seed*, which is an initial number the random function selects based on something unpredictable, like keystroke timing. For instance, you may have pressed your last key 684 milliseconds past the tick of the previous

second. For a random number between 0 and 9, your computer can grab the last digit of the 684 (which is a 4). The random seed determines what you'll get from your first `random()` call as well as all subsequent calls.

You can use Processing's `randomSeed()` function to set the random seed manually. Change the range argument to `10`, and insert a `randomSeed()` line at the very top of your working sketch:

```
randomSeed(213)
size(600, 250)
. . .
for i in range(10):
    point(random(width), random(height))
```

This `randomSeed()` function accepts a single argument, any integer of your choice, but you'll use `213` for this example. Unlike the 500-point (Figure 5-11) version in which no random seed had been defined, every run of the code produces the same pattern, on any computer that executes it.

This ability to ensure that the program generates the same sequence of pseudorandom numbers with every run is useful in many applications. For example, suppose you developed a platform game using levels composed of randomly positioned obstacles. Not having to place obstacles manually would save a lot of time. However, you discover that specific sequences of pseudorandom numbers produce more engaging levels than others. What's more, the resulting levels vary in difficulty, so you need to control the order in which the player progresses through them. If you're aware of the seed values that produce each level, you can reproduce any of them, on demand, with just an integer.

In the next section, you'll combine a `for` loop and the `random()` function to create interesting tile arrangements.

Truchet Tiles

Sébastien Truchet (1657–1729), a French Dominican priest, was active in the fields of mathematics, hydraulics, graphics, and typography. Among his many contributions, he developed a scheme for creating interesting patterns using tiles, which have since become known as *Truchet tiles.* The original Truchet tile is square and divided by a diagonal line between its opposing corners. This tile can be rotated in multiples of 90 degrees to produce four variants, as shown in Figure 5-12.

Figure 5-12: A Truchet tile, presented in its four possible orientations

These tiles are arranged on a square grid, either randomly or according to a pattern, to create aesthetically pleasing designs. Figure 5-13 presents just four possible arrangements, including a randomized tiling (bottom right) with some ordered approaches.

Figure 5-13: Four Truchet tile layouts

Next, you'll use the quarter-circle Truchet tile, shown in Figure 5-14, in its two possible orientations.

Let's apply the looping and randomness techniques you learned in this chapter to create different patterns using this tile. Create a new sketch and save it as *truchet_tiles*. Add the following setup code:

```
size(600, 600)
background('#004477')
noFill()
stroke('#FFFFFF')
strokeWeight(3)

for i in range(1, 145):
    arc(0, 0, 50, 50, 0, PI/2)
    arc(50, 50, 50, 50, PI, PI*1.5)
```

The new sketch has a blue background. Every shape you draw will have no fill and a white stroke of 3 pixels. This is for drawing the quarter-circle designs shown in Figure 5-14. Each tile is 50 × 50 pixels, so there's room for exactly 12 (600 ÷ 50) columns and 12 rows. Filling the display window, therefore, requires 144 (12 × 12) tiles, hence the range(1, 145).

Figure 5-14: Quarter-circle Truchet tiles

Run the sketch. A single tile should appear in the upper left corner (Figure 5-15).

Figure 5-15: All 144 tiles placed in the upper left corner

In actuality, in Figure 5-15, you're looking at all 144 tiles placed in the same position!

To control the column and row positioning, use col and row variables. Amend your script as per the boldface code:

```
. . .
col = 0
row = 0

for i in range(1, 145):
    arc(col, row, 50, 50, 0, PI/2)
    arc(col+50, row+50, 50, 50, PI, PI*1.5)
    col += 50
```

With each iteration of the loop, the `col` variable (tile y-coordinate) is increased by `50`. The result should be that each tile is placed to the right of its predecessor, as shown in Figure 5-16.

Figure 5-16: The remaining 132 tiles lie beyond the right edge.

There's a problem, though: the program doesn't know when to return to the left edge and begin a new row. Instead, the tiles overflow, extending out beyond the right edge where you cannot see them.

To correct this, nest an `if` statement within the loop:

```
. . .
for i in range(1, 145):
    . . .
    if i % 12 == 0:
        row += 50
        col = 0
```

The `i % 12` will return 0 for any value of `i` divisible by 12. In other words, if the remainder of a divide-by-12 operation is equal to 0, you know that you've just laid another 12 tiles. At this moment, the `row` variable is advanced by 50, and the `col` resets to 0. The next tile is now set up for placement at the beginning of a new row, which should result in a display window filled with tiles (Figure 5-17).
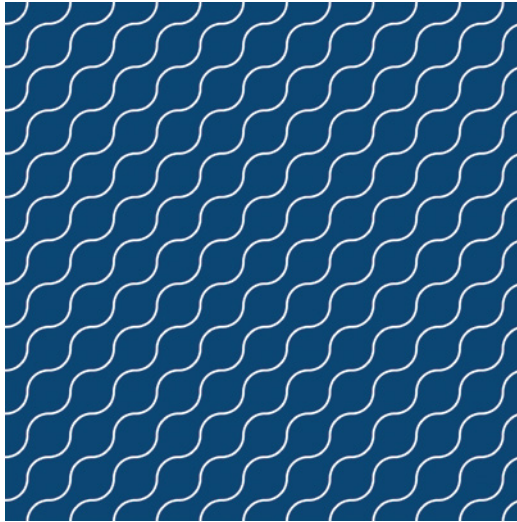
Figure 5-17: The display window filled with quarter-circle Truchet tiles

To make things more interesting, randomize the orientation of each tile by adding this `if...else` structure:

```
. . .
for i in range(1, 145):

    if int(random(2)❶):
        arc(col, row, 50, 50, 0, PI/2)
        arc(col+50, row+50, 50, 50, PI, PI*1.5)
❷   else:
        arc(col+50, row, 50, 50, PI/2, PI)
        arc(col, row+50, 50, 50, PI*1.5, 2*PI)

    col += 50
    . . .
```

A `random(2)` function ❶ will return a floating-point value ranging from 0 up to but not including 2. Converting the result to an integer by wrapping it in an `int()`, therefore, produces a 0 or 1. This is akin to flipping a coin, which is now performed with each iteration to decide which of the two tile orientations to pick. Because this "coin flip" operation returns a Boolean-compatible value—a 0 or 1—it can stand alone as the `if` statement's condition, no relational operators necessary. The `else` code ❷ runs if the result of the coin flip is a 0, because a 0 is equivalent to `False` (and `if` runs only on a `True`).

Each time you run the sketch, the display window presents a different pattern (Figure 5-18).
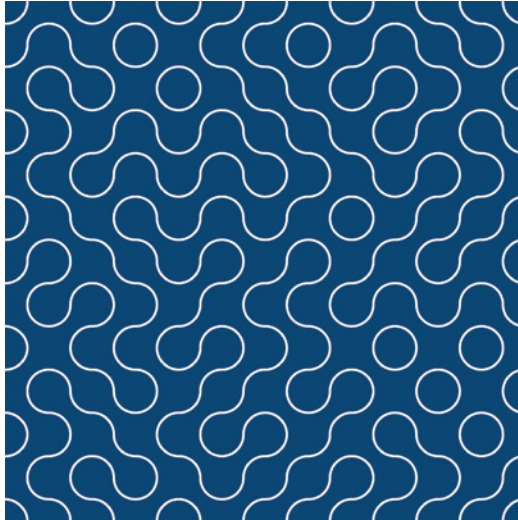
Figure 5-18: An arrangement of randomized quarter-circle Truchet tiles

If you've ever played the strategy game *Trax*, this pattern will look familiar. Another tile-based strategy game, *Tantrix*, uses a hexagonal adaptation of a Truchet tile. Of course, there's far more to tiles than the Truchet variety. You can try adding fills, switching out semicircles for diagonal lines, adding extra tiles to the set, or adding rules about which tiles can be placed next to one another (Figure 5-19). If you're looking for some fun projects, plenty of tiling patterns are available for inspiration.
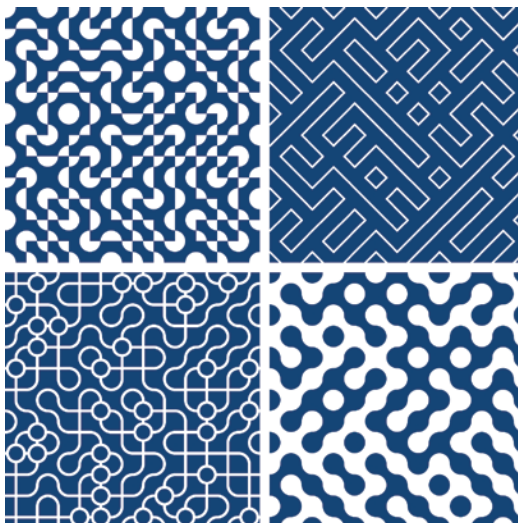
Figure 5-19: Variations of Truchet tiles

You can find code for some Truchet tile variations at *https://github.com/tabreturn/processing.py-book/tree/master/chapter-05-iteration_and_randomness/truchet_tiles_variations.*

As your programs grow more complex, you'll find multiple ways to code the same outcome. For example, you could have laid the quarter-circle Truchet tiles by using a loop within a loop, using `range()` functions with a step-size argument, in various combinations. Among the Truchet tile variations on Github, you'll find an example named *loop_within_a_loop* that uses this approach. Now that you understand control flow, you can begin thinking about how to optimize your algorithms for improved readability and efficiency.

Summary

You've now learned about iteration and how to program loops using `while` and `for` statements; this allows you to accomplish more in fewer lines of code, with code that's more adaptable. Loops will reappear throughout the course of this book, providing plenty more opportunities for you to master them.

This chapter also introduced randomness, which is useful in a variety of computing applications, including creative coding. The Processing `random()` function generates sequences of pseudorandom numbers, which you can control using a random seed in order to produce the same sequence of values each time you run your sketch.

The next chapter deals with motion. You'll learn how to add movement to your Processing sketches, and you'll also look at transformation functions as efficient ways to move, rotate, and scale your elements, which is especially useful for groups of shapes.

6

MOTION AND TRANSFORMATION

Applying movement to graphics of both living and inanimate objects instills them with character. Bouncy animation suggests playfulness; precise movement implies intensity, while slow motion can suggest heaviness. These techniques are applied in film, animation, dance choreography, and, of course, your favorite Pixar flick. But that's not all. Motion is prevalent in interface design, such as subtle button-hover effects or elaborate spinning graphics that appear while your content is loading.

In this chapter, you'll make things move by coding with motion and transformation functions. You'll learn how to manipulate the coordinate system with transformation functions, making it simpler to move, rotate, and scale your elements. In addition, you'll learn how to structure an animated Processing sketch by using the setup() and draw() functions. Motion literally adds a new dimension—time—to your Processing sketches.

Perceiving Motion

First, consider how motion is perceived. The brain is fed a snapshot from your retina many times each second. Provided that their screen can display a sequence of static images at a rate exceeding roughly 10 to 12 frames per second, the viewer will experience the illusion of smooth, flowing movement. Higher frame rates will appear even smoother.

Take a moment to note the two circles in Figure 6-1.
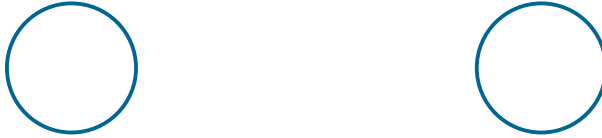
Figure 6-1: Two circles (positioned left and right)

If you displayed only the left circle for four seconds, followed by only the right circle for another four seconds, looping the sequence indefinitely (Figure 6-2), this would be an effective frame rate of 0.25 frames per second (or 0.25 *fps*). The result, most observers would agree, is a pair of alternating images depicting circles in two different positions.
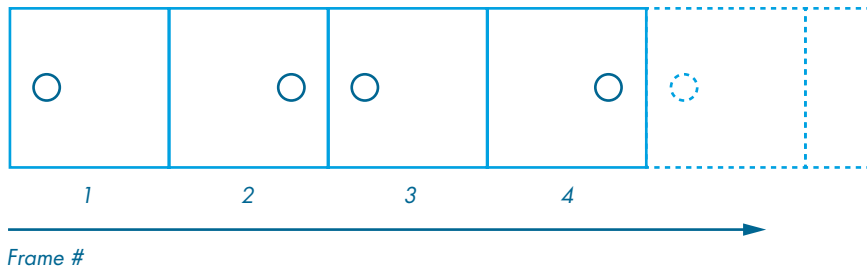
Frame #

Figure 6-2: Displaying alternating circles

However, speed up the frame rate to around 2.5 fps (10 times faster), and the observer will begin to interpret the sequence as a single circle bouncing between two points—as if the circle were moving across the gap in the middle. The illusion is referred to as *beta movement*. Increase the frame rate further, and the two circles will appear to flicker in sync with one another. From this experiment, you can see how frame rate doesn't affect only how fast or slow something moves, but also how you perceive the object's motion.

Note the numbering of the circles shown in Figure 6-3.

Now, suppose you want to animate this. Using the numbering to dictate the order, remove a single circle on each frame. On the first frame, remove just the circle labeled 0. On the second frame, replace circle 0 and remove only circle 1. Continue this process around the ring, and loop the animation indefinitely. Removing successive circles from each frame results in a gap that moves around the ring in a clockwise progression (Figure 6-4).
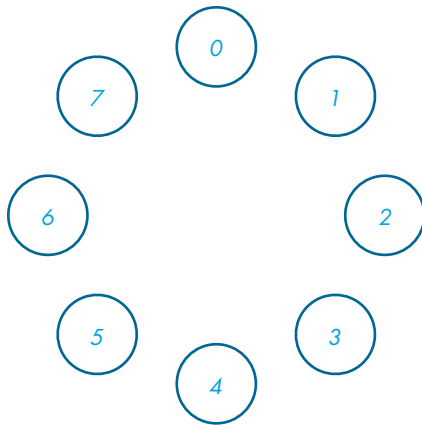
Figure 6-3: A ring of circles numbered in
a clockwise sequence
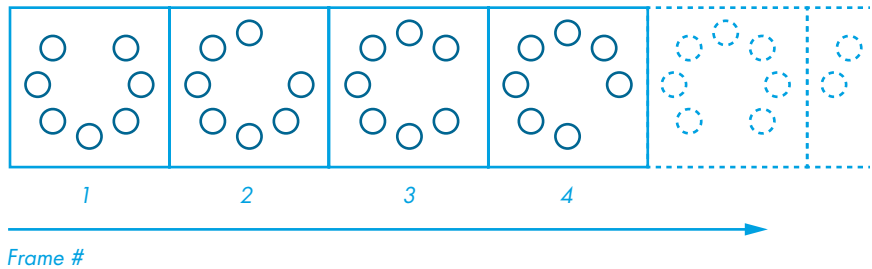
Figure 6-4: Animating the ring of circles

If you run the animation at 1 fps, the circle just ahead of a gap appears
to jump into the void left by the vacant circle (Figure 6-5).
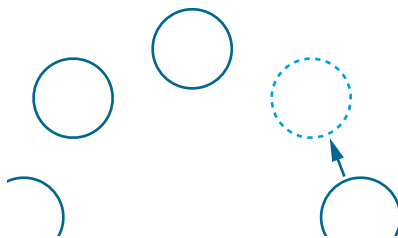
Figure 6-5: At 1 fps, the next circle seems
to leap into the gap.

At 25 fps, however, a rapidly moving phantom white dot seems to obscure
the circles beneath it as it races around the ring—an illusion called the *phi
phenomenon* (Figure 6-6).

Now you're ready to build a Processing sketch that, in addition to intro-
ducing Processing's animation functions, will allow you to experiment with
these phenomena.
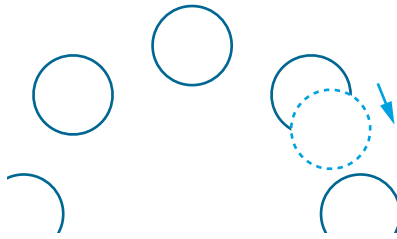
Figure 6-6: At 25 fps, a phantom white dot seems to obscure the circles.

Adding Motion to Processing Sketches

Processing gives you the option to draw to the display window a single time or multiple times over. For animation, you use the latter approach. To make an object move, you adjust its position with each frame drawn—and if you do it rapidly enough, in small enough increments, the result is smooth, flowing motion.

The draw() and setup() Functions

To make Processing draw something multiple times, you'll need to structure your code by using the setup() and draw() functions. Beneath those two functions, you can nest any of the functions or statements covered in the book so far. As indicated in Figure 6-7, where you place your code depends on when you want it to execute.

```
def setup():
    run this code once at the start

def draw():
    run this code every frame
```

Figure 6-7: Structuring code for motion

Any def keyword is followed by a function name, parentheses, and a colon. Chapter 9 covers def in more detail, but for now, just be aware that any code indented beneath a def belongs to that respective function.

The setup() code runs once at startup, and it typically includes things like your size() function and other lines that define your environmental properties. I'll get to draw() in more detail shortly, but first, create a new sketch, save it as *perceiving_motion*, and then add the following code:

```
def setup():
    size(500, 500)
    background('#004477')
    noFill()
    stroke('#FFFFFF')
    strokeWeight(3)
```

This code resembles just about every other sketch you've set up so far, except for the `def setup()` line. Whenever you intend to use a draw() function, you have to use setup() too. Now add the draw() function:

```
. . .
def draw():
    print(frameCount)
```

Processing invokes the code indented beneath the draw() function with each new frame. The `frameCount` is a system variable containing the number of frames displayed since starting the sketch. With each new frame, the draw() function calls the print() function, which in turn displays the current frame count in the console.

By default, draw() executes at approximately 60 fps. However, as the complexity of an animation increases, the frame rate is likely to drop as your computer struggles to accommodate the demands placed on it. Adjust the frame rate by using the `frameRate()` function (within the `setup()` block), and add a condition to draw() to print on even-numbered frames only:

```
def setup():
    . . .
    frameRate(2.5)

def draw():
    if frameCount % 2 == 0:
        print(frameCount)
```

With the `frameRate` set to `2.5`, the draw line runs two and a half times every second; this means that each frame is 400 milliseconds (0.4 of a second) in duration. Because the `print` line executes on every second frame, a new line appears in the console every 800 milliseconds (Figure 6-8).
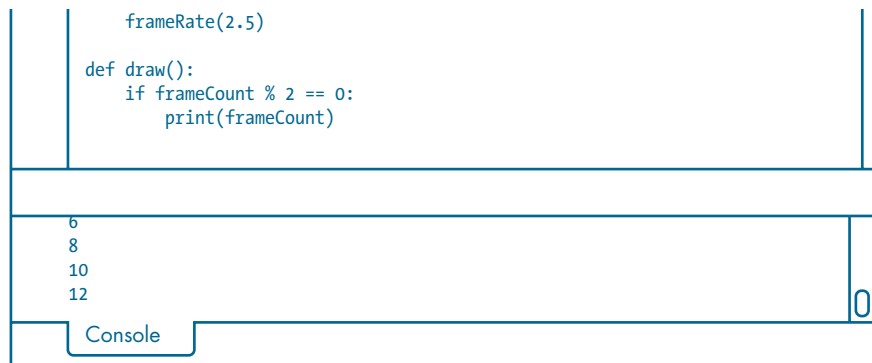
Figure 6-8: Printing the frame count on every even-numbered frame

To draw a circle on every even frame instead, use the following circle() line:

```
. . .
def draw():
    if frameCount % 2 == 0:
        circle(420, 250, 80)
```

Now run the sketch. You may be surprised to find that the circle does not flash on and off (Figure 6-9).
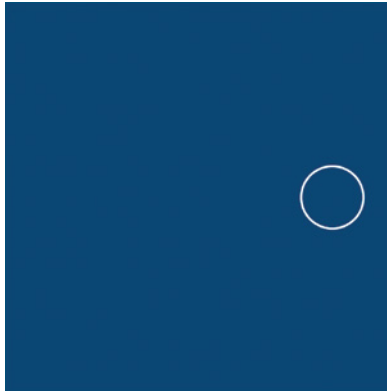
Figure 6-9: The circle does not "blink."

The reason the circle does not disappear on odd frames is that everything in Processing persists after it's drawn. On every even frame, the program draws another circle atop the existing "pile." The background() color within the setup() function runs once at the start, filling the display window in blue to form the bottommost layer of this persistent arrangement. To "wipe" each frame before drawing the next, you can redraw a background over everything.

Copy the background('#004477') line into the draw() section of your sketch:

```
. . .
def draw():
    background('#004477')
    if frameCount % 2 == 0:
        circle(420, 250, 80)
```

The new background() line clears every frame before it. Be sure you have placed it above the if statement. In most instances, a background() function will sit somewhere near the top of draw() to avoid clearing other shapes in the current frame.

Test the code. The result should be a blinking circle.

To recreate the ring of circles experiment from earlier (Figure 6-3), replace the existing `if` statement with a series of `if` statements:

```
. . .
def draw():
    background('#004477')
    hide = frameCount % 8 ❶

    if hide != 0:
        circle(250, 80, 80)
    if hide != 1:
        circle(370, 130, 80)
    if hide != 2:
        circle(420, 250, 80)
    if hide != 3:
        circle(370, 370, 80)
    if hide != 4:
        circle(250, 420, 80)
    if hide != 5:
        circle(130, 370, 80)
    if hide != 6:
        circle(80, 250, 80)
    if hide != 7:
        circle(130, 130, 80)
```

The current frame count is divided by 8 ❶, and the remainder is assigned to the `hide` variable. Each `if` statement will draw a separate circle provided it hasn't been flagged as the one to hide. For instance, on the 16th frame, `hide` is equal to `0` because 16 divides evenly by 8. On the 15th frame, `hide` is equal to `7` because 15 divided by 8 leaves a remainder of 7. On the 17th frame, `hide` is equal to `1`. The result is a stream of numbers that counts from 0 up to 7, then restarts at 0.

Run the sketch. Focus on the gap as it moves around the circle. At the current frame rate of 2.5 fps, the circle just ahead of a gap appears to jump into the void left by the vacant circle. But adjust the frame rate to 25 fps, and a phantom background-colored dot appears to obscure the circles beneath it as it races around the ring.

Global Variables

A *global variable* is one that you can access anywhere within your program. Up until this point in the book, almost every variable you have defined has been a global variable. You'll need to understand more about global variables to manage data across multiple frames.

Global variables are declared outside any function definitions (indented blocks beginning with `def`), usually somewhere near the top of your code. For instance, any variables that you declare outside `setup()` and `draw()` are automatically global. Conversely, any variables declared inside the indented lines of those two functions are accessible within that function alone.

As an example of this behavior in action, create a new sketch and save it as *global_variables*. Add the following code:

```
def setup():
❶  y = 1

def draw():
❷  print(y)
```

The y variable ❶ is declared within the setup() function. As such, y is accessible only within the indented lines of the setup() block. The y variable's scope, therefore, is considered to be *local* to setup(). *Scope*, in programming, deals with the regions where a variable (or other entity) may be accessed. In this instance, running the sketch produces an error (Figure 6-10), because you have attempted to access and print variable y from within the draw() function ❷.

Figure 6-10: The draw() function cannot access the y variable declared in setup().

Alternatively, you can move the y = 1 line outside the setup() function, which places it in the global scope; this permits either function to read it. Move this line to the top of your code and insert a pass statement in place of the location you moved it from:

```
y = 1

def setup():
    pass

def draw():
    print(y)
```

The draw() function has no problems accessing y now that it's declared outside setup(). The pass statement is a *null operation*—that is, nothing happens when it executes. You need to include a pass line because Python does not allow empty function definitions. This makes pass a useful placeholder for any code you have yet to write. Upon running the sketch, the console should print endless lines of 1s.

You can override the global y variable on a local level with another variable of the same name—in this case, another variable named y. Make the following adjustments to your code:

```
y = 1

def setup():
❶ y = 0
❷ print(y)

def draw():
❸ print(y)
```

The setup() function runs first—just once—and its print line ❷ displays a 0. This is because within the setup() function, you define the y as a 0 ❶. The outer (global) y is still equal to 1, and it's said to be *shadowed* by the setup()'s inner (local) y variable. The draw() code executes after the setup() code, and with every new frame, prints ❸ a 1 to the console. Run the sketch, quickly stop it, and then scroll up through the console output. The first line displayed is a 0; from there down, it's all 1s (Figure 6-11).


```
    y = 0
    print(y)

def draw():
    print(y)



0
1
1
...
Console
```

0
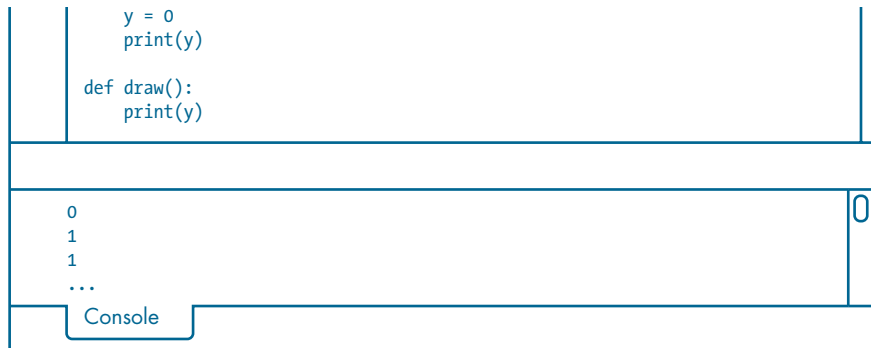
Figure 6-11: The global y variable is shadowed by the y = 0.

Next, remove the y = 0 line and add code that attempts to increment the global y variable by 1 with each frame:

```
. . .

def draw():
    y += 1
    print(y)
```

While you can read (or shadow) any global variable, writing or reassigning values requires additional code. As a result, this code should cause Processing to display an error (Figure 6-12).

```
        print(y)

    def draw():
        y += 1
        print(y)
```

UnboundLocalError: local variable 'y' referenced before assignment

```
1
processing.app.SketchException: UnboundLocalError: local variable
...
```

Console

Figure 6-12: The draw() function cannot reassign a value to y.

This is where the `global` statement is useful. Edit your code, inserting a `global y` line at the top of the `draw()` block:

```
. . .

def draw():
    global y
    y += 1
    print(y)
```

The global y variable is now bound to the local scope of `draw()`, and you may modify it as you wish. Run the sketch. The global y variable should now increment by 1 with each new frame (Figure 6-13).

```
    def draw():
        global y
        y += 1
        print(y)
```

```
1
2
...
```

Console

Figure 6-13: The global y variable is incremented by 1 with each new frame.

Global variables allow you to keep track of and update values between frames easily, which is especially useful for animating objects. Add a moving circle, the y-coordinate of which is controlled by the y variable:

```
y = 1

def setup():
    print(y)
    size(500, 500)
```

```
    noFill()
    stroke('#FFFFFF')
    strokeWeight(3)

def draw():
    . . .
    background('#004477')
    circle(height/2, y, 50)
```

I've placed the size, fill, and stroke properties in the setup() section of the code. Given that the stroke and fill are unchanged throughout the animation, there's no need to apply those properties repeatedly in draw(). The circle's y-coordinate, represented by variable y, moves the circle down as the frames advance. In Figure 6-14, a motion trail has been added to convey the direction of motion.
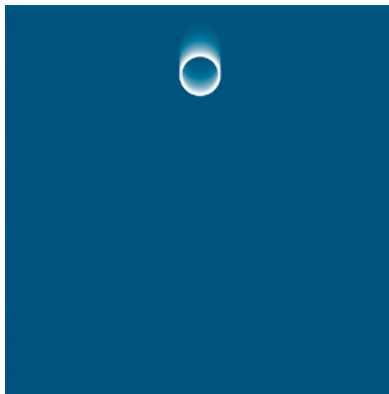
Figure 6-14: The circle moves down from the top of the display window.

When the circle reaches the bottom of the display window, it continues out of sight beyond the lower edge.

Saving Frames

Processing provides the saveFrame() function to save frames as image files. Whenever your sketch calls a saveFrame(), it saves a *Tagged Image File Format,* or TIFF, image in the sketch folder. You'll want to place this call at the end of your draw() function to ensure that you capture every shape rendered on the current frame. For instance, say you add the following code to a draw() function:

```
. . .
def draw():
    . . .
    if frameCount % 100 == 0:
        saveFrame()

    square(10, 10, 100)
```

As the animation encounters every 100th frame, a new image file appears in your sketch folder. This image file is named *screen-* followed by a four-digit frame count; where necessary, this frame count is padded with leading zeros, as shown in Figure 6-15. Because `saveFrame()` precedes the `square()` line, the square appears in every frame of the animation, but never in the saved image files.
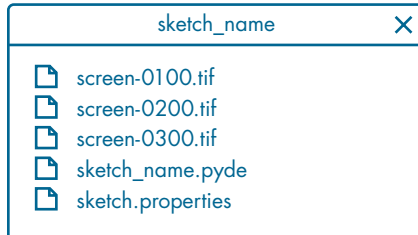
Figure 6-15: The saveFrame() function generates an image file named using the frame count.

If you want to save the file in an image format other than TIFF, such as JPG, PNG, or TARGA, include a filename argument with the relevant extension:

```
. . .
    saveFrame('frame.png')
```

In this case, you'd use the same filename for every image saved, which is okay for capturing a single frame, but will lead to overwriting when you call the same `saveFrame()` function multiple times. However, you can include a series of hash marks to make the frame count appear in the filename. This code generates a uniquely named PNG file with every save:

```
. . .
    saveFrame('frame-####.png')
```

Processing replaces the hash marks with the frame count and, if necessary, pads the count with leading zeros.

Challenge #5: DVD Screensaver

In this task, you'll combine `setup()`, `draw()`, global variables, and `if` statements to animate an object that bounces off the edges of the display window.

DVD players commonly feature a bouncing DVD logo as a screensaver (Figure 6-16), which appears after a given period of inactivity. You may have seen a variation of this on other devices, albeit with a different graphic. Intriguingly, people often find themselves staring at the pointless animation in the hope of witnessing the logo land perfectly in the corner of the screen.
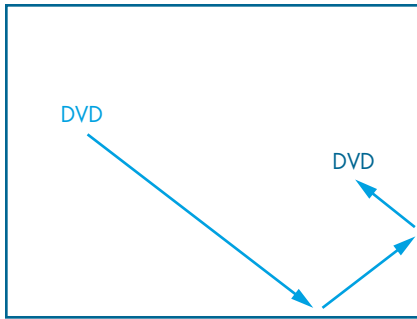
Figure 6-16: The logo bounces off edges
of the screen.

Create a new sketch and save it as *dvd_screensaver*. Add the following code:

```
y = 100
yspeed = 2

def setup():
    size(800, 600)
    fill('#0099FF')
    textSize(50)

def draw():
❶  global y, yspeed
    background('#000000')
❷  y += yspeed
    text('DVD', 100, y❸)
```

The code is similar to that of the previous example using the circle (Figure 6-14). In this instance, you include a yspeed variable. To use a single global statement for multiple variables, comma-separate them ❶. With each new frame, the program adds yspeed to the y variable ❷, which serves as the y-coordinate for the DVD text ❸. Upon running the sketch, the logo should move directly down (Figure 6-17), soon passing beyond the bottom edge of the display window.

Figure 6-17: The DVD text moves downward.

To make the logo rebound off the bottom edge of the display window, add the following `if` statement:

```
. . .

def draw():
    . . .
    if y > height:
        yspeed *= -1
```

When the `y` variable exceeds the `height` of the display window, the `yspeed` is multiplied by `-1`, sending the logo in the opposite direction. Run the sketch; the logo should rebound as it hits the bottom edge.

To move the logo diagonally, add some `x` values:

```
. . .
x = 100
xspeed = 2

. . .

def draw():
    global y, yspeed, x, xspeed
    background('#000000')
    y += yspeed
    x += xspeed
    text('DVD', x, y)
    . . .
```

Here, you've replicated everything you did with the `y` and `yspeed` variables for the `text()` function's `x` argument. The logo should now move vertically and horizontally. Run the sketch (Figure 6-18).

Figure 6-18: The diagonally moving DVD text rebounds near the lower-right corner.

When the logo rebounds off the bottom edge, the `yspeed` is inverted, but not the `xspeed`. This is the behavior you seek, but then the logo passes

through the right edge. Instead, the logo must rebound off every edge it encounters. Your challenge is to complete the task. If you need help, you can access the solution at *https://github.com/tabreturn/processing.py-book/tree/master/chapter-06-motion_and_transformation/dvd_screensaver/*.

Transformations

Processing's *transformation functions* provide convenient ways to manipulate elements by using translate, rotate, scale, and shear operations (Figure 6-19). You may apply transformations to individual shapes, groups of elements, or the entire drawing space.
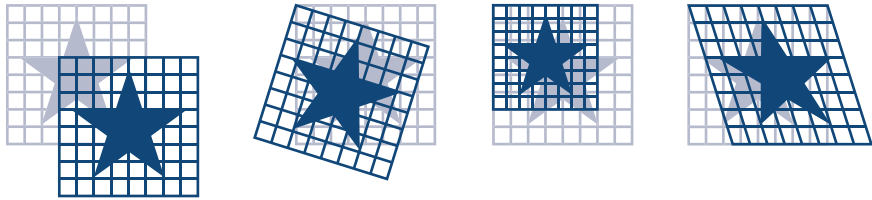
Figure 6-19: From left to right: translation, rotation, scaling, and shear transformations

Suppose you want to rotate a star shape (as shown in Figure 6-20) in a clockwise direction. This star is composed of vertices using a series of vertex() functions; an x-y coordinate pair defines the position of each vertex.
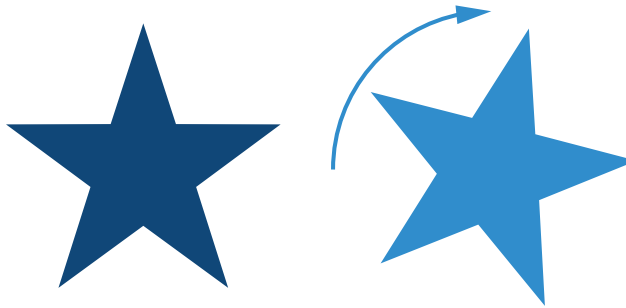
Figure 6-20: Rotating a star shape

Calculating the new positions of each vertex requires a *matrix*. You can think of a matrix as a table of numbers. For different transformations, you can add, subtract, or multiply each x-y coordinate pair with a *transformation matrix*. In the case of the star rotation, the matrix operation would look something like Figure 6-21. The *x* and *y* values in the square brackets labeled *vertex* represent the coordinate pair for a given vertex; this is multiplied by the *transformation matrix* to calculate a new rotated vertex position. The equation in the *result* brackets reveals the workings of the matrix math.

$$\underset{vertex}{\begin{bmatrix} x \\ y \end{bmatrix}} \times \underset{\substack{transformation \\ matrix}}{\begin{bmatrix} 0.9 & -0.3 \\ 0.3 & 0.9 \end{bmatrix}} = \underset{result}{\begin{bmatrix} (x \times 0.9) + (y \times -0.3) \\ (x \times 0.3) + (y \times 0.9) \end{bmatrix}}$$

Figure 6-21: A transformation matrix for rotation

If matrix math looks a little confusing, don't worry; Processing quietly handles all of it for you.

In the next section, you'll learn about the `translate()`, `rotate()`, `scale()`, `shearX()`, and `shearY()` functions. You'll also see how to use the `pushMatrix()` and `popMatrix()` functions for applying transformations to selected groups of elements.

Processing Transformation Functions

Create a new sketch and save it as *transformation_functions*. Within the sketch's folder, create a *data* subfolder and then follow these steps:

1. Open your web browser and go to *https://github.com/tabreturn/processing.py-book/*.
2. Navigate to *chapter-06-motion_and_transformation*.
3. Download the *grid.png* and *grid-overlay.png* files.
4. Place both files in your *data* subfolder.

Add the following setup code:

```
size(800, 800)
noFill()
noStroke()
grid = loadImage('grid.png')
image(grid, 0, 0)
grido = loadImage('grid-overlay.png')
```

The `grid` variable and `image()` lines load and display the *grid.png* graphic. The *grid-overlay.png* file is loaded into the variable `grido`, but it's not rendered in the display window yet; you'll display it later in this task.

translate()

The `translate()` function accepts two arguments: an x-offset and y-offset. Ordinarily, an x-y coordinate of (0, 0) marks the upper left corner of the display window. This point is called the *origin*. Using `translate()`, you can reposition the coordinate system, which shifts the origin and influences everything you draw after that.

Add a `translate()` function to your *transformation_functions* code, and display the grid-overlay graphic by using a new `image()` line.

```
. . .
translate(150, 100)
image(grido, 0, 0)
```

The `translate()` function moves the entire coordinate system 150 pixels across and 100 pixels down. The `image()` function draws the grid-overlay graphic—a pale blue version of the first grid image—at (0, 0). The *grid-overlay.png* graphic has a transparent background, so you should see the *grid.png* file showing through it. Run the sketch to confirm that the output matches Figure 6-22.
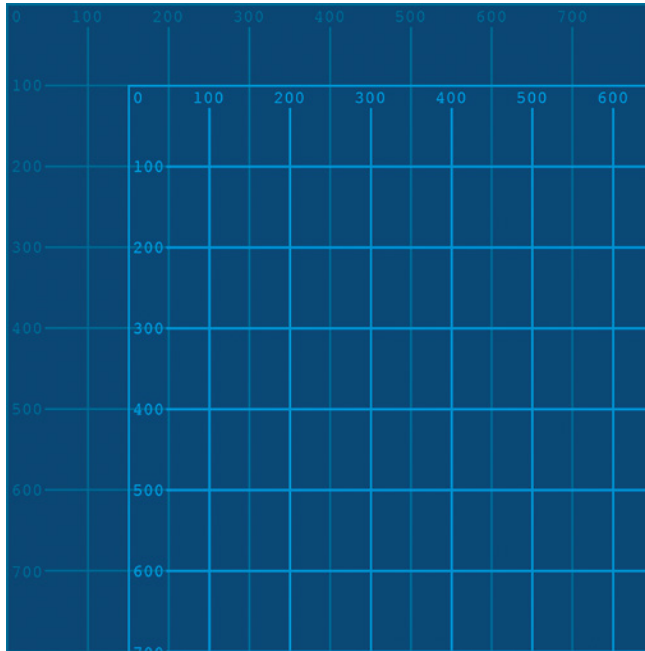
Figure 6-22: The grid image with the translated grid-overlay displayed above it

The x-y coordinate (0, 0) no longer aligns with the upper left corner of the display window. The grid-overlay graphic serves as a visual representation of your new, shifted coordinate system.

Add a red and a yellow square:

```
. . .
fill('#FF0000')
square(0, 0, 100)

fill('#FFFF00')
square(100, 0, 100)
```

The red and yellow squares share a y-argument of 0, but the yellow square has an x-coordinate of 100. Run the sketch. Processing positions both squares relative to your new origin. The yellow square should appear to the right of the red (Figure 6-23).

Transformations are cumulative, meaning that each subsequent transformation uses the current coordinate system as a reference, so you could have placed the yellow square 100 pixels to the right by using an additional `translate(100, 0)`:

```
. . .
translate(100, 0)
fill('#FFFF00')
square(0, 0, 100)
```

The new `translate()` line has an x argument of 100, and the x argument for the `square()` is now 0. The visual result should be the same as Figure 6-23.
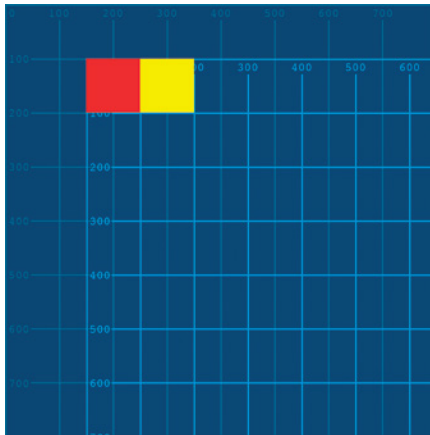
Figure 6-23: Horizontally adjacent red and yellow squares

TRANSFORMATIONS WITHIN DRAW()

Transformations within the `draw()` block are reset every time it re-executes. While you may have several `translate()` and/or other transformation functions within your `draw()` block, the effects will not carry over into the next frame.

For cumulative transformations across frames, you can use global variables as transformation arguments. In this way, the values can increment with each frame.

In Chapter 5, you learned how to use a loop to arrange Truchet tiles. A row and column variable kept track of where to place the tiles. Alternatively, you could have used translate(), moving the coordinate system with each iteration of the loop.

rotate()

The rotate() function rotates the coordinate system around its origin (0, 0). It accepts a single argument specified in radians. Positive values rotate clockwise, and negative values rotate counterclockwise. As with all transformation functions, the effect is cumulative. Moreover, you can mix rotate() and other transformation functions as you please.

Add a new rotate() line beneath your first translate() function to rotate the grid-overlay graphic and red and yellow squares:

```
. . .
translate(150, 100)
rotate(QUARTER_PI)
. . .
```

The rotate() function uses an argument of QUARTER_PI radians, equivalent to 45 degrees. Note that QUARTER_PI is a predefined Processing variable, equivalent to writing PI/4.

Run the sketch. The two squares should appear to be rotated as a group, along with the grid-overlay graphic (Figure 6-24).
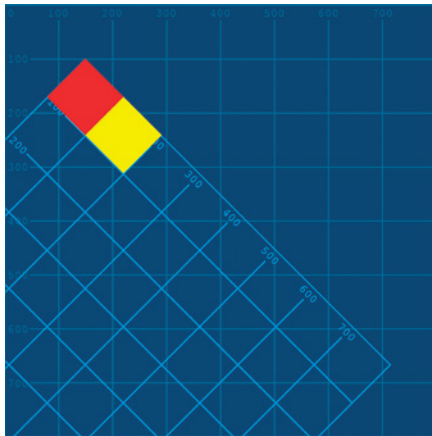
Figure 6-24: Rotating the grid-overlay graphic and two squares

The coordinate system is rotated around the current origin, which serves as the pivot point. Recall that this origin has been offset by 150 pixels for x and 100 pixels for y by the translate() function.

The order of functions matters. For instance, switching the translate() and rotate() lines produces different visual results. Figure 6-25 provides a

comparison. The ghosted squares depict the result of the transformation that occurred first. The right image is produced by performing the `rotate()` first, when the origin is aligned with the upper left corner of the display window.
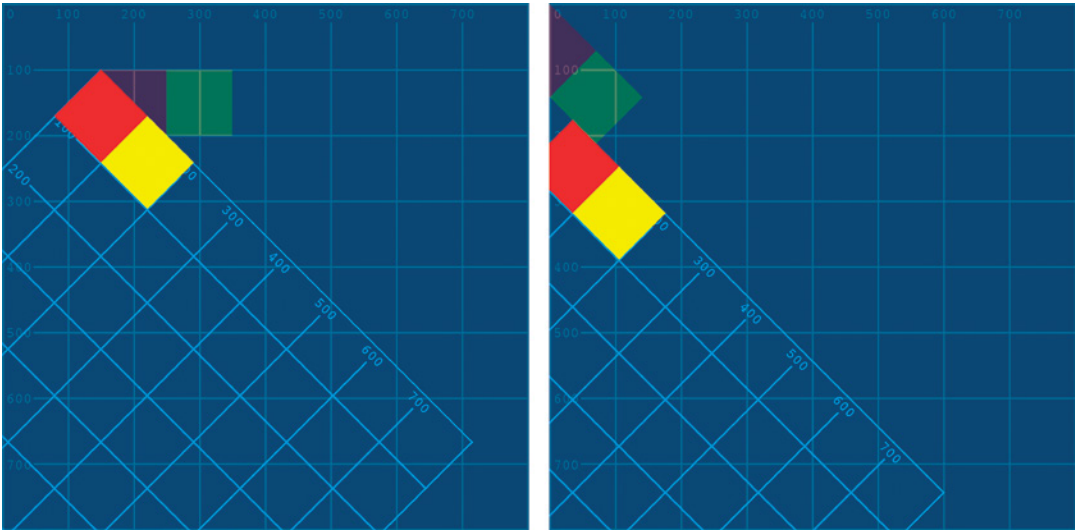
Figure 6-25: The order of the `translate()` and `rotate()` functions matters; the image on the left shows `translate()` first, and the image on the right shows `rotate()` first.

To rotate a square around its center, as opposed to its upper left corner, align the center of the square with the origin by offsetting the x and y arguments for `square()`.

scale()

The `scale()` function resizes the coordinate system. One argument will scale proportionately; two arguments control the x-scale and y-scale. A `scale(1)` or `scale(1, 1)` will have no effect, as those are the default scale values.

To decrease the scale, use a floating-point value between 0 and 1. Reduce the size of your existing elements:

```
. . .
translate(150, 100)
rotate(QUARTER_PI)
scale(0.5)
. . .
```

The scale value of `0.5` scales the elements to half their original size. Just as with `rotate()`, the scaling is relative to the origin of the current coordinate system. In other words, (0, 0) stays in place, and everything shrinks toward this point (Figure 6-26).
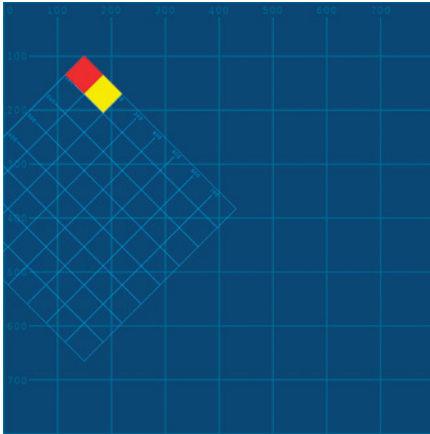
Figure 6-26: Halving the size by using `scale(0.5)`

Any value above 1 scales upward. For instance, to double the size of everything, use `scale(2)`. To reflect/flip on a given axis, use a negative value. For example, `scale(-1, 1)` flips everything horizontally, producing a mirror image of your elements.

shearX() and shearY()

Shearing a shape skews it along the horizontal or vertical axis. The result is a distorted shape with the same area. A typical shear example is transforming a rectangle into a parallelogram with slanted sides.

The `shearX()` and `shearY()` functions apply a horizontal and vertical shear, respectively. Each function accepts a single argument specified in radians.

To apply a vertical shear to your grid-overlay graphic and two squares, comment out the `rotate()` line and apply a 45-degree vertical shear by using a `shearY()` function:

```
. . .
translate(150, 100)
#rotate(QUARTER_PI)
scale(0.5)
shearY(QUARTER_PI)
. . .
```

The `rotate()` function is commented to make the direction of the shear more visually apparent. The `shearY()` argument is a positive number, so the shear is applied in a clockwise direction. Figure 6-27 contrasts the result of these code changes (left image) and a `shearX()` operation (right).

You now know how to combine transformation functions; however, you'll often want to contain the transformation effects to a limited selection of elements. Next, let's look at how to use multiple coordinate matrices within a single sketch.
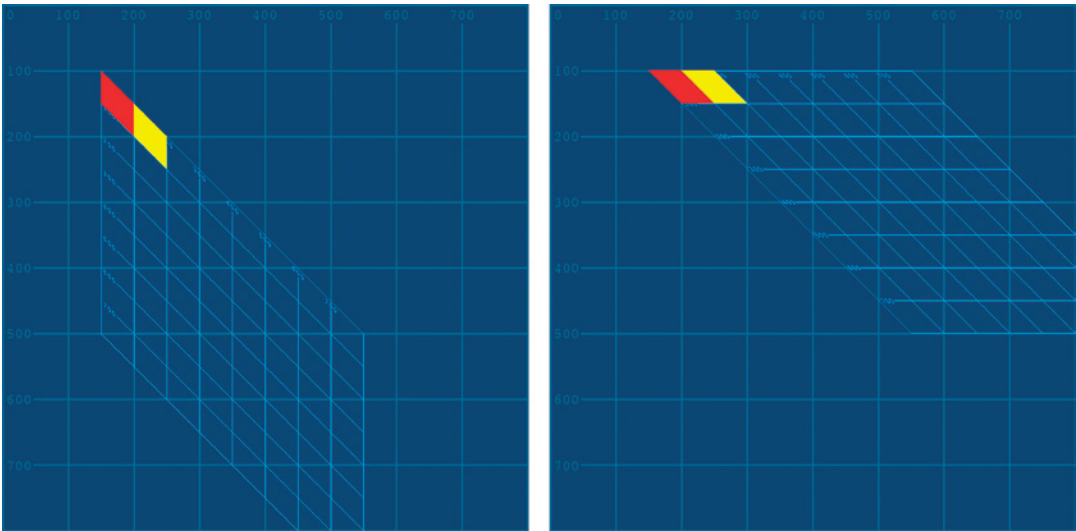
Figure 6-27: shearY(QUARTER_PI) (left) and shearX(QUARTER_PI) (right)

pushMatrix() and popMatrix()

The pushMatrix() and popMatrix() functions allow you to isolate the effects of any transformation functions. In this way, you can perform different transformations on selected elements, which is especially useful for groups of elements.

Any elements you add to your sketch are positioned relative to the coordinate system's origin. Recall that each new transformation function affects the position or orientation of the origin and that each new transformation is influenced by any that precede it.

RESETTING THE COORDINATE SYSTEM WITH RESETMATRIX()

If you need to clear all of your transformations, you can use the resetMatrix() function to replace the current matrix with the *identity matrix*, which resets to the default coordinate system. For example, add the resetMatrix() line just before the yellow square to clear all of the transformations preceding it:

```
. . .
resetMatrix()
fill('#FFFF00')
square(0, 0, 100)
. . .
```

The yellow square is now rendered relative to the original origin (upper left corner of the display window), at the standard scale, without any of the rotation or shear effects (Figure 6-28).
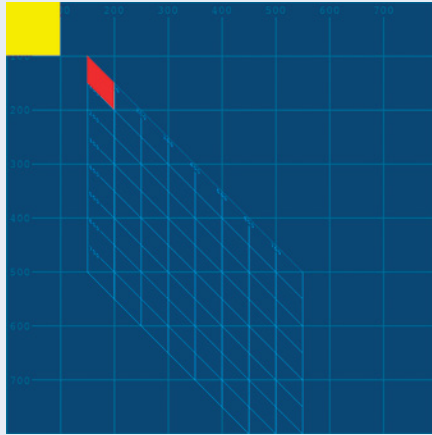
Figure 6-28: The yellow square code is preceded by resetMatrix().

Remove the resetMatrix() line before you continue.

If you want to apply translate() and scale() to the yellow square, but not shearY(), isolate the red and yellow squares, placing each within pushMatrix() and popMatrix():

```
      . . .
❶  translate(150, 100)
   #rotate(QUARTER_PI)
   scale(0.5)

   pushMatrix()
❷  shearY(QUARTER_PI)
   image(grido, 0, 0)
   fill('#FF0000')
   square(0, 0, 100)
   popMatrix()

   pushMatrix()
❸  translate(100, 0)
   image(grido, 0, 0)
   fill('#FFFF00')
   square(0, 0, 100)
   popMatrix()
```

The pushMatrix() functions create new matrices for shearY() ❷ and translate() ❸, which both extend upon the translate(150, 100) above ❶. The popMatrix() function restores the coordinate system before the previous pushMatrix() line. I've added another grid-overlay graphic to help visualize what is happening with the two coordinate systems.

As an alternative, you could undo the shear by adding `shearY(-QUARTER_PI)` after the red square, but pushing and popping matrices is the more elegant solution.

Run the sketch. As shown in Figure 6-29, the yellow square should appear translated and scaled, but not sheared.
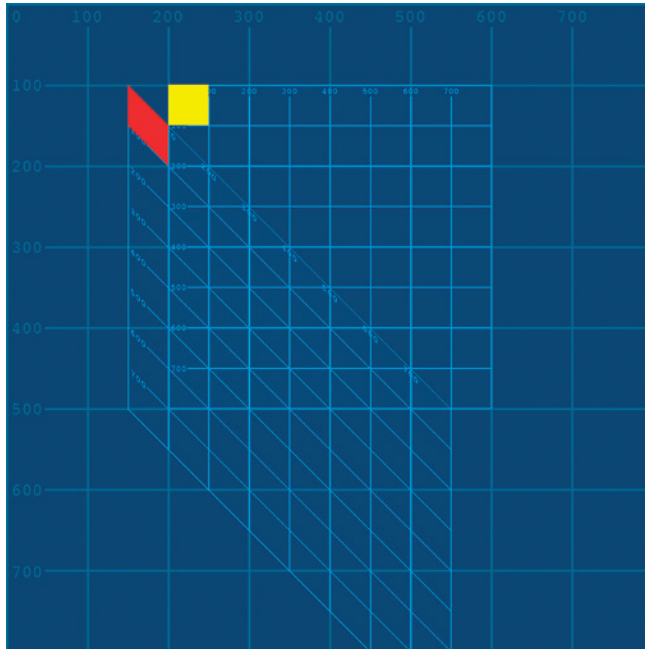
Figure 6-29: The yellow square is translated and scaled, but not sheared.

Now imagine that you want to move drawings made of multiple shapes across the display window. Figure 6-30 depicts a fish tank simulation; each fish is composed of many shapes. Each eye (a circle) has its own x-y coordinate, and so does every vertex that defines a curve or straight line.
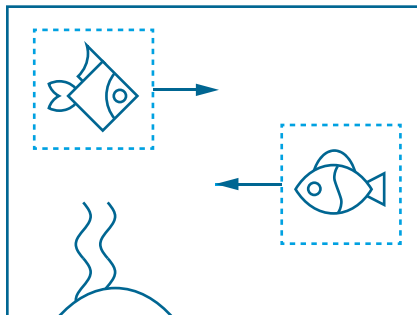
Figure 6-30: Translating groups of shapes by using pushMatrix() and popMatrix()

To track and update all of these x-y coordinates, you must store them in global variables to increment with each frame. The more efficient approach is to define each fish within a pair of `pushMatrix()` and `popMatrix()` functions. In this way, you can control the position of one fish by using one global x-y coordinate pair and a `translate()` function.

Experiment with the `pushMatrix()` and `popMatrix()` functions containing different groups of shapes, each employing a different sequence of transformation functions. You can add animation if you like. Reuse the `image(grido, 0, 0)` line within each of your `pushMatrix()`... `popMatrix()` stacks to help visualize what's happening.

Challenge #6: Analog Clock

In this challenge, you'll use all the techniques you've learned in this chapter to create an analog clock that displays the current time. The clock will update every second, so you'll need to use `draw()`. To rotate the second, minute, and hour hands, you'll use transformation functions.

Create a new sketch and save it as *analog_clock*. Add the following code:

```
def setup():
    size(600, 600)
    frameRate(1)
    noFill()
    stroke('#FFFFFF')

def draw():
    background('#004477')
```

The frame rate is set to 1, enough to update the second hand's position each second.

To retrieve the relevant time values, use the Processing `hour()`, `minute()`, and `second()` functions. Each function communicates with your computer clock to return an integer value; these functions require no arguments. Add code to the `draw` block that displays the current time in the console:

```
    . . .
    h = hour()
    m = minute()
    s = second()
    print('{}:{}:{}'.format(h, m, s))
```

Run the sketch. With each new frame, your console displays the current hours (0 to 23), minutes (0 to 59), and seconds (0 to 59), separated by colons. The time should match that of your system clock, usually displayed in the corner of your screen.

Creating a digital-style clock (that is, no hands, just numbers) in Processing is a simple matter of combining time and `text()` functions. For an analog clock, however, you need to convert the hours, minutes, and seconds into angles of rotation.

Begin your clock by drawing the face and hour hand:

```
. . .
❶ translate(width/2, height/2)
  strokeWeight(3)
❷ circle(0, 0, 350)

  # hour hand
❸ strokeWeight(10)
  line(0, 0, 100❹, 0)
```

The translate() function ❶ positions the origin in the center of the display window. This will make rotating the clock hands simpler, because the rotate() function rotates around the origin of the coordinate system. The circle() function, with its x-y arguments both set at zero ❷, is centered in the display window (Figure 6-31). The hour hand is the thickest (and shortest), with a stroke weight of 10 ❸ and length of 100 pixels ❹.

Figure 6-31: A clock face with an hour hand

The hour hand currently rests along 0 radians (pointing east). Recall that when drawing using the arc() function, the angle opens from this point, clockwise (southward). However, your clock will be offset by three hours should the hand begin from a three o'clock position. Calibrate this using a rotate() function:

```
. . .
rotate(-HALF_PI)
# hour hand
. . .
```

The HALF_PI is equivalent to PI / 2; by prepending this with a - sign, you rotate counterclockwise. Run the sketch. The hour hand should now point to twelve o'clock (directly upward).

The next step is to calculate how many radians the hand advances with each hour. Consider that a complete rotation is 2π radians; therefore, one hour equals PI * 2 / 12. So, six o'clock is PI * 2 / 12 * 6. Rather than writing PI * 2, though, you can use TAU. For example, six o'clock is equal to TAU / 12 * 6.

Rotate your hour hand to the current hour:

```
. . .
# hour hand
rotate(TAU / 12 * h)
. . .
```

At twelve o'clock, the hour hand points directly upward. This is because TAU / 12 * 12 is equal to TAU, or one complete rotation. For every other hour, the hand should point to the correct position (Figure 6-32). Of course, the angle of the hand will depend on what time of day it is.

Now add the minute and second hands. The final result should look something like Figure 6-33.

The second hand should advance each second. Compare the time in the console to the visual output to ensure that your code is working correctly. If you need help, you can access the solution at *https://github.com/tabreturn/ processing.py-book/tree/master/chapter-06-motion_and_transformation/analog_clock/.*
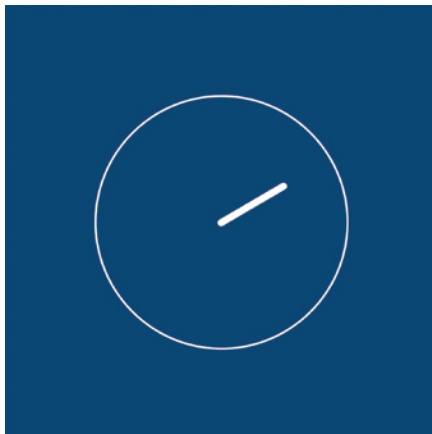
Figure 6-32: The hour hand pointing to two o'clock

Figure 6-33: The completed clock

Summary

In this chapter, you learned how to structure a Processing sketch for animation. To manage variables between frames, you learned how to use global variables. You can increment global variables every frame to control shape coordinates for smooth animation. You also now know how to save frames as images. You might save an animation as a sequence of images so that you can combine them into a movie by using video editing software.

You also saw how transformation functions manipulate the coordinate system, allowing you to translate, rotate, scale, and shear your elements. And you learned to modify the coordinate system to apply transformations to a select group of elements. It's far easier to move a group of shapes by using a single translate operation than to manage a large number of coordinate variables. Moreover, applying rotate, scale, and shear operations to a single shape, let alone group, would otherwise involve complex matrix calculations.

In the next chapter, you'll learn about Python lists and how to read in data from external files. Lists will unlock powerful ways to manage and manipulate values as collections of elements, rather than individually. To help visualize list values, you'll also explore data visualization techniques.

7

WORKING WITH LISTS AND READING DATA

When you need to work with multiple values, you can group them into a single variable by using a Python list. The *list* data type stores any number of items in collections you can manage and manipulate dynamically and efficiently. For example, you could create a list to store the titles of your favorite movies and use built-in methods to insert new favorites, reorder the rankings, or display only titles ranked between 30 and 40.

In this chapter, you'll learn to create and manipulate lists, and then you'll combine them with loops to access and perform actions with each item. In keeping with this book's visual theme, you'll generate graphical representations of list data, including a chart that displays brightness and RGB mixtures for a list of colors and another that plots the bestselling video games of all time. You'll see how to adjust list values to affect visual output, observing how the charts adapt to changing data.

You'll also learn how to read in data from text files and how text-based formats differ from other file formats. You'll move your Python list data into CSV-formatted text files and load it in when your sketch runs, allowing you to prepare data with other tools, such as spreadsheets.

Introducing Lists

Lists hold multiple values that are related or belong together. For example, consider programming a video game in which players wander about collecting various objects—keys, weapons, armor upgrades, and so forth— to advance to a new level. Your game needs to track those items, which you can store in an inventory list.

To denote a list, use square brackets and separate each element with a comma. As an example, here's a simple list for some game items:

```
inventory = ['key', 'gem', 'sword', 'apple', 'book']
```

This list contains five strings and is assigned to a variable named `inventory`.

Performing repetitive operations on collections of items is a common programming challenge. Suppose you want to display a grid with all the objects a player has collected (Figure 7-1). You can write a `loop` statement to access each item in the inventory and draw it in a cell. If the size of the list changes—because the player has added or dropped items—the loop will adapt, so you can write the code once and then have the program fill the appropriate number of cells to depict the inventory items.

Figure 7-1: A player inventory from the game Minetest

In Python, a single list can contain any mix of data types and duplicate values. For example, this top-score entry stores multiple types of data:

```
topscore = ['LEO', 54120]
```

The player name LEO is a string, and the high score is an integer.

Lists can include as many elements as you want, and you can even define an empty list by using just a pair of square brackets with nothing within them, which is useful if you intend to add items while your program is running.

Lists are ordered, and ordering is significant in many situations—for example, in this sequence of rainbow colors:

```
rainbow = [
  'red',
  'orange',
  'yellow',
  'green',
  'blue',
  'indigo',
  'violet',
]
```

When defining a list, you can write it across multiple lines, as shown here, to make your code easier to read and edit. Python also permits an optional trailing comma after the last element. Having this extra comma can help when you want to add or shuffle list values; just be careful not to forget a comma where necessary.

Creating and Accessing Lists

To familiarize yourself with defining, accessing, and modifying lists, create a new sketch. Save it as *rainbow_list* and add the following code:

```
rainbow = ['blue', 'orange', 'yellow']
print(rainbow)
```

For now, this rainbow is missing a few colors, and the sequence is incorrect, so you'll use various list operations to add and shuffle colors as you progress through this section. First, run the code to verify the following console output:

```
['blue', 'orange', 'yellow']
```

Printing the rainbow variable displays all three values, complete with square brackets and commas.

In many instances, you'll want to retrieve an individual element instead of a whole list. To display a given color, specify its position, or *index*, in square brackets. Note that Python list indices start at 0, so to print the first element, enter the following:

```
. . .
print(rainbow[0])
```

Run the sketch to confirm that the console displays blue.

The second element, orange, has an index of 1, and the last element in this list, yellow, has an index of 2. To print items 1 and 2, enter the following (note that throughout this chapter, the comments alongside the print lines indicate what should appear in your console):

```
. . .
print(rainbow[1])    # displays: orange
print(rainbow[2])    # displays: yellow
```

This syntax may remind you of slice notation from working with strings in Chapter 3, and it should, because it works the same way. Just as with slice notation, use -1 to access the last element of the list, and extract a subset of elements by using a range defined with a colon. Try the following code:

```
. . .
print(rainbow[-1])    # displays: yellow
print(rainbow[-2])    # displays: orange
print(rainbow[0:2])   # displays: ['blue', 'orange']
```

If you specify any index beyond the bounds of the list, such as rainbow[3] or higher, Processing will display an IndexError message.

Modifying Lists

Lists can be dynamic in behavior, changing while your program runs. You can overwrite any element with a new value and use different list methods to insert new elements or remove existing ones. For a game inventory, you might replace a weapon if a player finds a more powerful one, and add or remove elements as the player trades items.

Returning to the rainbow example, you need to replace blue with red as the first color in the rainbow list. To modify an existing list element, reassign it a new value as you would any other variable, but with lists, you need to specify the element index in square brackets. Add the following line to the end of your *rainbow_list* sketch:

```
. . .
rainbow[0] = 'red'
```

The red string now replaces blue, overwriting it as the first item in the list. Printing the rainbow list should confirm this:

```
print(rainbow)        # ['red', 'orange', 'yellow']
```

Blue is no longer in the rainbow list.

Let's look at several of the most useful list methods, along with code to add to your working sketch. Each example builds on the code before it, so work through all of them sequentially, entering the lines as you progress.

The append() Method

The `append()` method adds an element to the very end of a list, whatever its length. Add blue to the end of the `rainbow` list:

```
rainbow.append('blue')
print(rainbow)        # red, orange, yellow, blue
```

Note that the comments after the `print()` function in these examples contain only the sequence of colors; when you actually print the list, the console will display `['red', ...,'blue']` with all of the brackets and quotation marks.

The extend() Method

To add all the elements in one list to the end of another, use the `extend()` method:

```
colors = ['indigo', 'violet']
rainbow.extend(colors)
print(rainbow)        # red, orange, yellow, blue, indigo, violet
```

The `colors` list, which contains indigo and violet, is now added to the original `rainbow` list.

The index() Method

The `index()` method returns the index (the position in the list as an integer) for any element that matches the argument provided. If there are multiple matches, this method detects the first instance. Use an argument of `'yellow'` to test this:

```
yellowindex = rainbow.index('yellow')
print(yellowindex)   # 2
```

Try different color arguments. If no matching value exists, Processing displays a `ValueError` message.

The insert() Method

The `insert()` method accepts two arguments: the first is the index at which to insert the element; the second is the value. Insert green into the middle of the list with an index argument of `3`:

```
rainbow.insert(3, 'green')
print(rainbow)        # red, orange, yellow, green, blue, . . .
```

Green is now in the position that blue used to occupy, shifting blue one index higher along with every color to the right of it.

The pop() Method

The pop() method accepts a single argument: the index of an element to remove. The "popped" value is returned should you need to use it for another operation. Pop indigo from the list and assign it to a variable named i; then print i and rainbow to confirm that your console output matches the comments shown here:

```
i = rainbow.pop(5)
print(i)            # indigo
print(rainbow)      # red, orange, yellow, green, blue, violet
```

If you aren't concerned with using the popped value, remove the i = part. Now, use pop() with no argument to remove the last item in the list:

```
i = rainbow.pop()
print(rainbow)      # red, orange, yellow, green, blue
```

The console output should confirm that violet is removed from the list.

The remove() Method

The remove() method removes the first element with a value that matches the argument provided. Re-add indigo and violet by using the extend() method, and then remove indigo with the remove() method:

```
rainbow.extend(colors)
print(rainbow)      # red, orange, yellow, green, blue, indigo, violet
rainbow.remove('indigo')
print(rainbow)      # red, orange, yellow, green, blue, violet
```

After extending the list, rainbow is back to a seven-color list. After the remove line, the list is down to six colors again, with no indigo.

Python provides other list methods, but these should suffice for you to start manipulating lists. Any decent Python reference or internet search should cover the rest. For example, if you want to reorder list elements, look up the reverse() and alphanumerical sort() methods. The Processing reference also includes several list methods, which are standard Python (as opposed to Processing) features, and they are functional in any Python environment.

Combining Loops and Lists

You can program loops to work on lists, potentially saving countless lines of manual instruction. As an example, say you want to create a *Breakout*-style game (Figure 7-2). In this type of game, the player controls the paddle at the bottom of the screen with the goal of bouncing the ball upward to destroy all the bricks. You could create a list to store the bricks so that when

the player hits a brick with the ball, that brick would be removed from the list. In some levels, you could have additional bricks appear during play, which would mean you'd need to insert new list elements.

Figure 7-2: LBreakout2, an open source Breakout clone

You've likely played a variant of this game and are probably aware that, upon destruction, certain bricks drop power-ups. You also know that the bricks come in different colors, and that some may be invisible but solid, while others may take multiple hits to destroy. You can program all of those additional properties by using lists of lists. Lists can contain other lists, which, in turn, can contain further nested lists (see "Creating Lists of Lists" on page 144).

If your list is named `bricks` and contains the fills for 60 bricks, rendering each brick would require at least as many lines of code as you have elements. For instance, you might use the following code to draw each brick with a `rect()` function:

```
bricks = [
  '#FF0000',
  '#FF0000',
  . . .

# brick A1
fill(bricks[0])
rect(0, 0, 30, 10)
# brick A2
```

```
fill(bricks[1])
rect(30, 0, 30, 10)
. . .
# brick F10
fill(bricks[59])
rect(270, 50, 30, 10)
```

Notice that every brick rendered requires a fill() and rect() function. Even if you remove the comments, that's 120 lines (60 × 2) of code to draw the complete list. This is hardly efficient, nor can the code handle a list that might fluctuate in length.

Drawing Shapes by Using a List of Color Values

For this exercise, you'll draw a rainbow-colored sequence of bands from a list of hexadecimal values, beginning with a single band using a fill() and rect() function. You'll then adapt the code to use a loop that draws the entire list. To begin, add the following code to your *rainbow_list* sketch:

```
. . .
size(500, 500)
noStroke()
background('#004477')

bands = [
  '#FF0000',  # red
  '#FF9900',  # orange
  '#FFFF00',  # yellow
  '#00FF00',  # green
  '#0099FF',  # blue
  '#6633FF'   # violet
]

# red band
translate(0, 100)
fill(bands[0])
rect(0, 0, width, 50)
```

Up until this point, the sketch has relied exclusively on the console for output. This code begins by defining a display window size, no stroke, and a background color. The bands list holds hexadecimal values for a six-color rainbow with comments to identify each color value. The first (red) band is drawn using translate(), fill(), and rect() functions. Run the sketch. The result should be a single, horizontal red band on a blue background (Figure 7-3).

Figure 7-3: The result of running the sketch is a single red band.

You've drawn the first band in the list, and the next step is to adapt the code to use a `for` loop that draws all six bands.

When you combine a `for` loop with a list, Python assigns each successive list value to the loop variable, using the length of the list to determine the number of iterations required. To make your program draw every band in the `bands` list, comment out the existing `fill()` and `rect()` functions, and then add a loop that draws the complete rainbow for you:

```
. . .
#fill(bands[0])
#rect(0, 0, width, 50)

for band❶ in bands:
    fill(band)
    rect(0, 0, width, 50)
    translate(0, 50❷)
```

In this instance, the code is easier to understand if you name the loop variable `band` ❶ instead of something like `i`. The `band` variable is equal to `'#FF0000'` on the first iteration, `'#FF9900'` on the second, and so forth. A `translate()` function moves the coordinate system down the height of a band ❷. With each iteration, Processing applies the next fill in the list and draws a new rectangle below the last one. The result is a stack of six rainbow-colored bands that span the width of the display window (Figure 7-4). Note that the green band will be brighter on a computer screen than it is in a printed book. Standard printing inks (cyan, magenta, yellow, and key/black—CMYK) cannot replicate the intensity of the shades of green on a digital display.

Figure 7-4: A rainbow sequence of six color bands

In this example, Python retrieves each element in the list, so you don't need to specify any index values. In the next section, you'll use the `enumerate()` function to retrieve the index for each element as well as the value.

SIX-COLOR RAINBOWS?

Hold on! What happened to indigo? According to Wictionary.org, indigo is a "purplish-blue color," and violet is a "blueish-purple color." So, why is there indigo and violet, but no purple band in a rainbow?

Purple is a combination of two spectral colors. There is no wavelength of purple light; it exists only as a combination of red and blue waves. Violet, however, is an actual spectral color with its own wavelength of approximately 380 to 420 nanometers. Indigo lies somewhere between blue and violet, but exactly where, if at all, is a matter for debate. In his famous prism experiments, Isaac Newton defined seven rainbow colors, squeezing in indigo just before violet. You may wonder, why seven colors from a blended array spanning the visible spectrum? It's because the number seven has occult significance. It's no coincidence that there are seven colors in the rainbow, seven days of the week, and seven musical notes that make up the Western major scale. Today, however, color scientists are inclined to divide the spectrum at blue and violet, leaving no room for indigo.

Pink Floyd's iconic *The Dark Side of the Moon* album cover depicts a prism that splits a white beam into an array of rainbow-like bands. Have you ever counted the color bands in this design?

In this book, we'll drop indigo in favor of the six-color rainbow.

Looping with enumerate()

For some looping tasks, you need each element's index and value. For instance, say you have an ordered list of your favorite movies and want to print each title alongside its rank (the index). You can do so with the enumerate() function.

To use the enumerate() function to get the index of each color band in your rainbow, provide two variable names between the for and in. These two variables will hold your index and a corresponding value, respectively, for any iteration. Modify the code in your *rainbow_list* sketch:

```
. . .
#for band in bands:
❶ for i, band in enumerate(bands):
    fill(band)
    rect(0, 0, width, 50)
  ❷ fill('#FFFFFF')
    textSize(25)
    text(i, 20, 35)
    translate(0, 50)
```

The i and band variables represent the index and fill value, respectively ❶. The extra fill and the two text lines below it draw index numbers over each rectangle ❷.

Run the sketch. You should now see a white number in each band (Figure 7-5), although the 2 doesn't show up particularly well over the yellow.
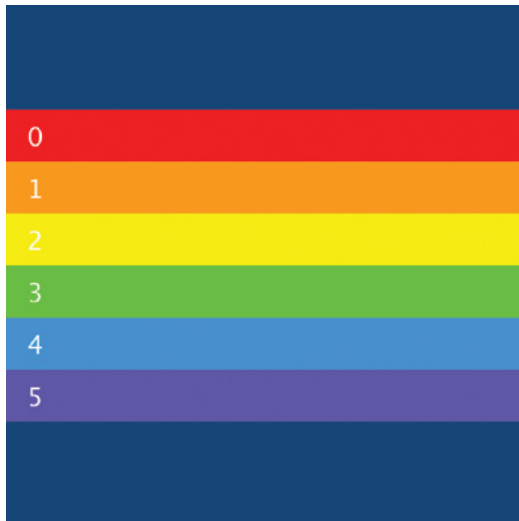
Figure 7-5: A numbered sequence of rainbow bands

Use an enumerate() function wherever you need to work with list indices or keep count of loop iterations. For any other loop operations on lists, a plain for loop should suffice.

Creating Lists of Lists

Although the concept of having lists within lists may seem complicated, appropriately nested lists make complex datasets easier to manage. In this practical data visualization task, you'll create a variation of a bar chart. This chart will measure the relative brightness of six colors. Figure 7-6 shows a simplified representation of what you're working toward. Notice that yellow, the brightest color, has the longest bar.
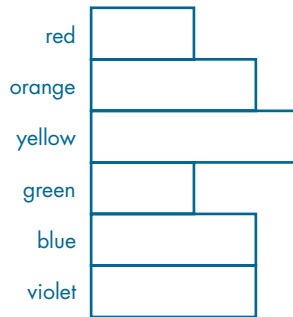
Figure 7-6: A simplified, outlined representation of the bar chart

The final chart will include color, and the bars will be further divided into segments of red, green, and blue to represent the RGB mixture of each color (more on this later).

DATA VISUALIZATION

Data visualization is the graphical representation of data using charts, graphs, maps, and other diagrams. This topic relates neatly to many coding concepts, and it makes for some intriguing and enlightening visual output. A good example is Frederic Brodbeck's *Cinemetrics* project (see "What Is Creative Coding?" on page xviii) that analyzes DVD movie data to generate visual fingerprints of films. For many inspiring data visualizations, you can peruse the collection of works showcased at *https://informationisbeautiful.net/*. When writing Processing code, you're no longer limited to whatever your spreadsheet software can conjure. Instead, you can explore novel ways to visualize data, ranging from highly abstract or playful to highly informative to anything in between.

The first step in creating the bar chart is to start a new sketch and save it as *lists_of_lists*. Add the following setup code:

```
size(500, 380)
background('#004477')
noFill()
stroke('#FFFFFF')
```

```
strokeWeight(3)

h = 50
translate(100, 40)

bands = 6
rect(0, 0, 40, h*bands)
```

The h variable defines the bar height, and the `translate()` function defines the upper left corner. The visual result should appear as a vertical bar; this represents a total number of six bands (Figure 7-7). The height of the bar represents a single integer value: 6. If `bands` were equal to 7, the rectangle that defines the bar would extend beyond the bottom of the display window.

Figure 7-7: A bar 6 × 50 pixels tall

The next step is to split the existing bar into six segments, which will later form the horizontal bars. Add a new `bands1` list of rainbow colors to the end of your sketch, along with a loop that draws a rectangle using each color:

```
. . .
bands1 = [
  '#FF0000',
  '#FF9900',
  '#FFFF00',
  '#00FF00',
  '#0099FF',
  '#6633FF'
]

for band in bands1:
    fill(band)
    rect(0, 0, 40, h)
    translate(0, h)
```

This `bands1` list contains a series of six hexadecimal color values. These define the fills for each segment. The `for` loop draws the rainbow-colored segments in a column arrangement that conceals the first bar (Figure 7-8).
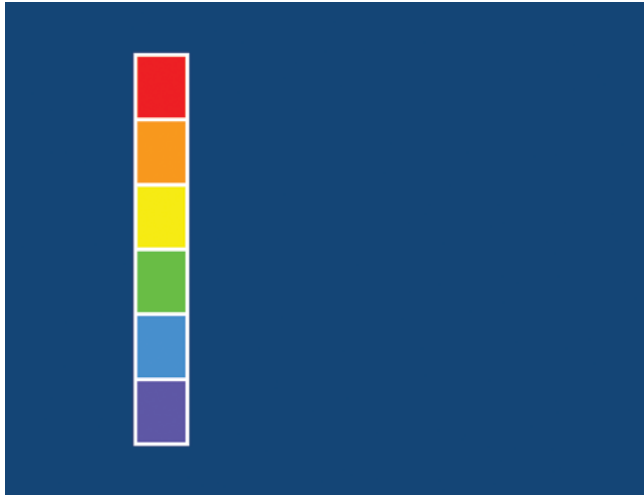
Figure 7-8: Rainbow-colored rectangles placed over the original bar

The next step is to extend each block of color toward the right to form horizontal bars. The width of each bar will be determined by the brightness of its respective color. To calculate brightness, add together the red, green, and blue values that make up any color. For example, consider white. It's the brightest "color" on your screen; it's represented in hexadecimal as #FFFFFF, and if converted to percentages, expressed as 100 percent red, 100 percent green, and 100 percent blue. That's an overall brightness of 300 percent, or if you prefer to average it out, it's 300 ÷ 3 = 100 percent bright.

To manage the colors as RGB percentages, you'll need an integer value for each R/G/B primary, as opposed to a single hexadecimal string value. Add a new `bands2` list to the end of your code, wherein each element contains a list of three integers representing the red/green/blue mix of each color:

```
· · ·
bands2 = [
  [100, 0, 0],
  [100, 60, 0],
  [100, 100, 0],
  [0, 100, 0],
  [0, 60, 100],
  [40, 20, 100]
]
```

To access any list element within another list element directly, include a second pair of square brackets. For example, to retrieve the percentage of green in the second (orange) band, enter the following:

```
print(bands2[1][1])  # 60
```

In this case, the green value is 60, which you can confirm in the console.

To work with the percentages in the bands2 list, set colorMode() to use RGB values between 0 and 100. To draw the bars, reset and translate the coordinate system, and then add a loop that draws rectangles filled in with various shades of gray:

```
. . .
colorMode(RGB, 100)
resetMatrix()
translate(100, 40)

for band❶ in bands2:
    r = band[0]
    g = band[1]
    b = band[2]
  ❷ sum = r + g + b
  ❸ avg = sum / 3
  ❹ fill(avg, avg, avg)
    rect(0, 0, sum❺, h)
    translate(0, h)
```

With each iteration, band is assigned the next list of RGB percentage values ❶. These values are added together ❷, averaged to calculate a brightness value ❸, and the bar fill is set to a shade of gray using equal quantities of red/green/blue based on this average ❹. The brightness value also determines the width of the bar ❺. Run the sketch to view the result (Figure 7-9).

Oddly, the green bar (fourth from the top) is indicated as equivalent in brightness to the red (top) bar. Recall also that the green is even brighter on your screen than in print. The math is correct, but the human eye has a greater number of green receptors, making us more sensitive to green light, so the green band appears brighter. There are ways to compensate for this mathematically. If you'd like to test it out, you can multiply the r, g, b variables using the following values:

```
    . . .
    r = band[0] * 0.64
    g = band[1] * 2.15
    b = band[2] * 0.22
    . . .
```
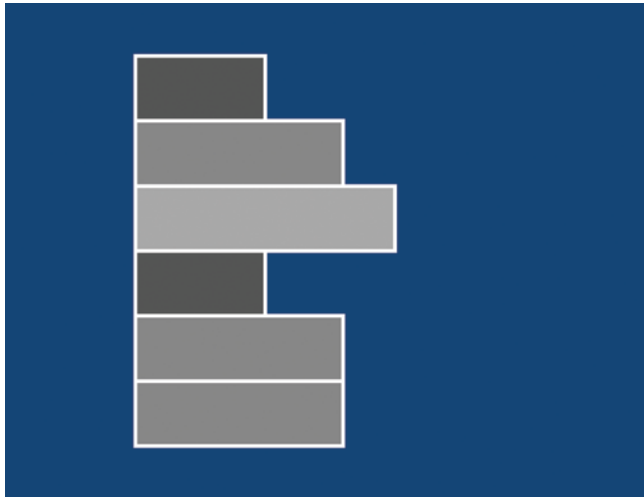
Figure 7-9: The widths of each bar represent the relative brightness
of each color.

Now, the yellow bar (third from the top) is the only bar wider/brighter
than the green one. For this task, however, I want to work with the averaging
formula, so remove any multipliers to revert to the averaged values.

NOTE   *This list-of-lists structure is called a* two-dimensional list. *You might even refer
to the list of hexadecimal values (*bands1*) as a* one-dimensional list, *but it's less
common to hear that term unless programmers are contrasting one type with the
other. List structures actually reflect the dimensionality of the data. So, adding
data to* bands1 *affects the y-axis; with* bands2, *the data controls both the x- and
y-axes (a two-dimensional system).*

Next, adapt the existing loop so that each bar indicates the different
quantities of primary color that make up its fill:

```
    . . .
    r = band[0]
    g = band[1]
    b = band[2]
    #sum = r + g + b
    #avg = sum / 3
    #fill(avg, avg, avg)
    #rect(0, 0, sum, h)
❶   fill('#FF0000')
    rect(0, 0, r, h)
❷   fill('#00FF00')
    rect(r, 0, g, h)
❸   fill('#0099FF')
    rect(r+g, 0, b, h)
    translate(0, h)
```

The `rect()` functions form horizontal bars containing up to three segments each. The size and fill of each segment are governed by how much red ❶, green ❷, and blue ❸ the color band contains. Even with the `colorMode()` set to RGB, Processing can still interpret fill arguments in quotes as hexadecimal.

Run the sketch to view the result (Figure 7-10). Red, the top bar, is mixed using nothing but red. Violet, the bottom bar, is predominantly blue, but also contains some red and a little green.
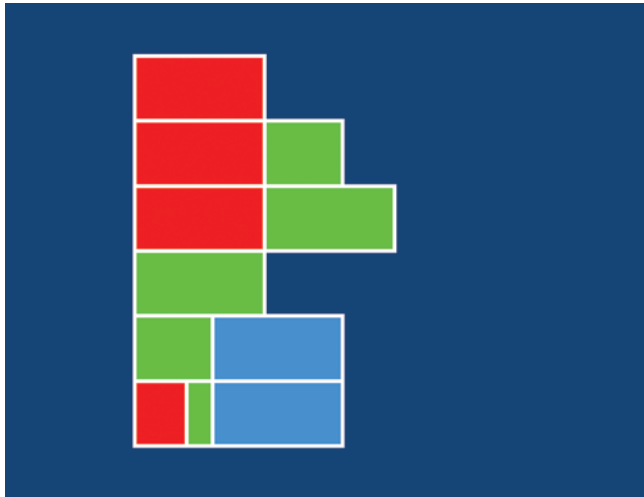
Figure 7-10: Each bar displays its proportion of RGB primaries.

If you show the chart to others, they likely will have no idea what color each bar represents, so adding labels will help elucidate matters. Add a label element to each band:

```
. . .
bands2 = [
  [100, 0, 0, 'red'],
  [100, 60, 0, 'orange'],
  [100, 100, 0, 'yellow'],
  [0, 100, 0, 'green'],
  [0, 60, 100, 'blue'],
  [40, 20, 100, 'violet']
]
. . .
```

Then, add some lines to your loop to draw each label:

```
. . .
for band in bands2:

    . . .
    fill('#FFFFFF')
    textAlign(RIGHT)
    text(band[3], -20, 30)
    translate(0, h)
```

This sets the text fill to white, right-aligns it, and writes a color label alongside the bar. Run the code to view the result (Figure 7-11).

Figure 7-11: Completed graph with labels

Many lists work just fine with a single dimension, such as shopping lists. You can think of two-dimensional lists as grids or tables, which makes them useful for plotting 2D graphics. Three-dimensional and other higher-dimensional lists have their places, but before employing such a structure, consider whether adding another position to your two-dimensional list may be more sensible.

Challenge #7: Breakout Level

In this challenge, you'll recreate a *Breakout* level. The setup code will include a three-dimensional list. Working with such a list requires a nested loop—that is, a loop inside another loop.

The result should look like Figure 7-12. Note that you're not creating a playable game with working inputs; it's more like a screenshot grabbed during play.

Create a new sketch and save it as *breakout_level*. Add the following code to draw the ball and paddle:

```
size(600, 600)
noStroke()
background('#000000')

# ball and paddle
fill('#FFFFFF')
circle(350, 440, 18)
rect(300, 520, 190, 40)
```
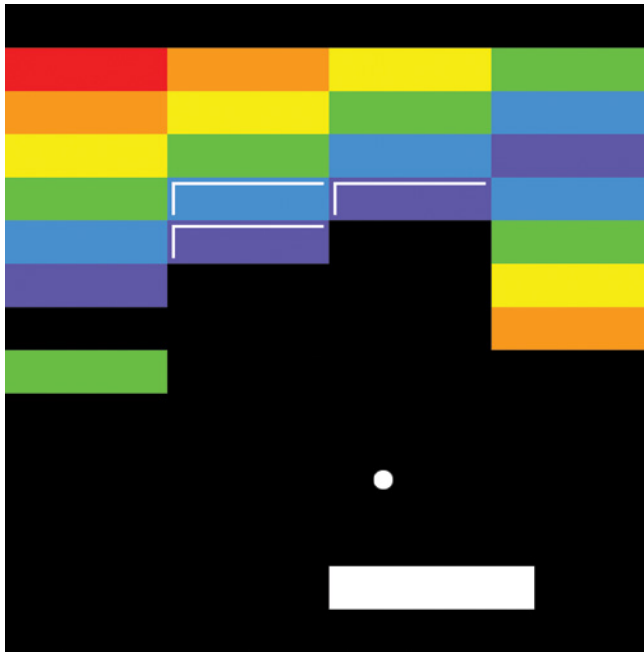
Figure 7-12: Completed Breakout task

This code should render an empty black stage with the white ball and paddle, but no bricks yet.

Now add the data for the bricks. To save time, copy and paste the code from my GitHub repository:

1. Open your browser and go to *https://github.com/tabreturn/ processing.py-book/*.
2. Navigate to *chapter-07-working_with_lists_and_reading_data.*
3. Locate and open the *bricks.txt* file.
4. Copy and paste the contents of *bricks.txt* into your sketch.

Here's the code if you'd prefer to type it in:

```
r = '#FF0000'  # red
o = '#FF9900'  # orange
y = '#FFFF00'  # yellow
g = '#00FF00'  # green
b = '#0099FF'  # blue
p = '#6633FF'  # violet

bricks = [
  # col 0  col 1  col 2  col 3
  [ [r,1], [o,1], [y,1], [g,1] ],  # row 0
  [ [o,1], [y,1], [g,1], [b,1] ],  # row 1
  [ [y,1], [g,1], [b,1], [p,1] ],  # row 2
  [ [g,1], [b,2], [p,2], [b,1] ],  # row 3
  [ [b,1], [p,2], [   ], [g,1] ],  # row 4
```

```
  [ [p,1], [    ], [    ], [y,1] ],  # row 5
  [ [    ], [    ], [    ], [o,1] ],  # row 6
  [ [g,1], [    ], [    ], [    ] ]   # row 7
]
```

To make this more readable, I've entered the bricks list in a way that reflects the visual positioning of each brick. In the following order, each brick has a fill color and hit count (indicating the number of hits required to destroy it). I represent each missing brick by using an empty list.

Take the first brick as an example: [r,1]. This brick has a fill of red and requires one (remaining) hit to destroy. You can infer the column and row positions from the lists in which the brick resides; in this case, it's row 0, column 0. Add two print() statements to confirm this information:

```
. . .
❶ print(bricks[0])          # displays row 0 items
❷ print(bricks[0][0])       # displays the very first brick
```

These print lines display the first element in bricks, a list of the four bricks that make up row 0 ❶, and the first brick in row 0 ❷. If you want to retrieve the color of the first brick, enter the following:

```
print(bricks[0][0][0])  # displays #FF0000
```

Note that the color variable r holds a hexadecimal value, so what you see in the console is the hexadecimal value for red.

As I mentioned previously, you'll need to employ a nested loop for this task. The following lines will help you get started:

```
. . .
bw = width / 4
bh = height / 15
translate(0, bh)

for row in bricks:

    for col, brick in enumerate(row):

        if len(brick):
            # code to draw a brick
            x = col * bw
```

The bw variable defines a brick width based on fitting four columns into the display window; bh calculates the brick height. The outer for loops through the rows; the inner for loops through the bricks within each row. The col and brick variables hold the column number and brick, respectively. You use the len() function to determine whether this brick is a placeholder (an empty list). A brick with a length of 0 is equivalent to False, and Python skips the x = col * bw line. The x variable will hold the x-coordinate to draw each brick. Complete the task to match the result shown in Figure 7-12. Note that the bricks located roughly in the center

have a hit count of 2 and must include a shine effect. If you need help, you can access the code at *https://github.com/tabreturn/processing.py-book/tree/master/chapter-07-working_with_lists_and_reading_data/breakout_level/*.

In the next section, you'll learn how to work with data from external files, and you'll use list techniques with Processing functions that read in the contents of text files.

Reading Data

Python—and by extension, Processing—can handle many types of file data. For instance, you could use Processing to create a game that incorporates various audio and video files, storing these multimedia assets in your *data* subfolder. You've loaded image data from PNG files into your Processing sketches in previous chapters; this section focuses on loading data stored in text-based files.

You've also worked with values stored in lists, but using Python's list syntax to retype data from other sources can be tedious, especially for large and swappable datasets. An alternative is to manage and prepare data outside Processing by using something like a spreadsheet, save it in a text-based format, and then read in the file contents when you run your sketch. To understand what separates text-based files from other files, and how you might use them to store data, let's start with a brief introduction to file formats.

File Formats

A *file format* is a standardized means of encoding information for storage on a digital medium. Many formats exist, and each is interpreted differently. For example, applications are encoded in *executable formats*, such as Android Package Kit (APK) files for Android or executable (EXE) files for Windows. Some *multimedia formats* include MPEG-1/2 Audio Layer III (MP3) for music or JPG for images.

You can identify a file's format by its *file extension*. File extensions typically comprise three letters, always preceded by a dot, and tacked onto the end of a filename. To simplify user interaction, many operating systems hide file extensions, but if you dig around in your Windows File Explorer or Mac Finder settings, you can make your file manager show the extensions. Your system relies on these file extensions to open files with the appropriate app and to display icons or generate thumbnails (Figure 7-13).
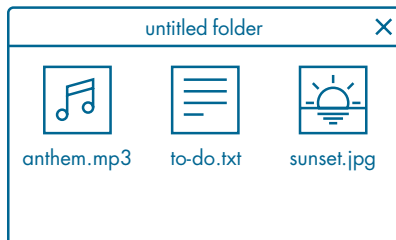
Figure 7-13: A file manager in icon view with file extensions revealed

When you remove or rename a file extension, this association is lost. Perhaps you've tried to open an MP3 file in a text editor and gotten a bunch of garbled characters, something like this:

```
· · ·
���:����zc��E9���yoO��F�;#C��@##�&�#�##HV�D��#���X���#�&
2XNf�##M�#�#���#J��,8,#`}##�#�4R�f�#E��V���d@��P������G��r
jS#gbx�:P+�A��'��Q�IF��5�0�i.�A���sG�P"����oA~�#
· · ·
```

Text editors are designed for editing text-encoded files; therefore, they attempt to interpret the audio data as characters. Although you might be able to spot some intelligible metadata in there somewhere, it's 99 percent gobbledygook. If you open this same file with iTunes, Windows Media Player, or VLC, you'll hear music.

Some file formats are text based, which means you can open them in any text or code editor and make some sense of the content. To clarify, by *text based*, I mean *plaintext*, not a Microsoft Word document with fonts of varying colors and sizes in bold and italic. You may be wondering why people even use plaintext, but it's appropriate for simple to-do lists and writing just about any programming language, Python included. For instance, Processing files are plaintext, albeit with a *.pyde* file extension.

CSV

Comma-separated values (CSV) files, which have the *.csv* extension, provide a simple approach to formatting plaintext data. You'll download a CSV that contains a track list of Pink Floyd's album *The Dark Side of the Moon*.

Each line of a CSV file is one entry, and each entry consists of one or more fields separated by commas. Here's an abridged track listing of *The Dark Side of the Moon* in CSV format:

```
location,title,creator,album,trackNum
file:///music/SpeakToMe.mp3,Speak to Me,Pink Floyd,The Dark Side of the Moon,1
file:///music/Breathe.mp3,Breathe,Pink Floyd,The Dark Side of the Moon,2
· · ·
```

The first line of this file contains the field headings, and the following lines provide the details of each track. Your spreadsheet software (Microsoft Excel, LibreOffice Calc, or similar) will associate itself with any files bearing the *.csv* extension. Opening any CSV file in a spreadsheet displays the information in the typical row-and-column arrangement (Figure 7-14). This is useful for preparing CSV data, but be aware that none of the styling (cell sizes, font colors, and so on) is retained once you save back to CSV.

Figure 7-14: The full playlist.csv *file open in LibreOffice Calc*

> **NOTE** *CSV files don't always rely on a comma to delimit each field. For instance, tab- and space-separated values are common as well.*

You'll now write code that loads the track-list data from a CSV file. Create a new sketch named *csv* with a *data* subfolder and complete the following steps:

1. Open your browser and go to *https://github.com/tabreturn/processing.py-book/*.
2. Navigate to *chapter-07-working_with_lists_and_reading_data*.
3. Download the *data.zip* file.
4. Extract the ZIP archive, and move *playlist.csv* to the sketch *data* subfolder.

Processing provides the loadStrings() function to read in text-based files. It accepts a single argument (a path) that points to your text file and returns the contents as a list of strings, each element representing a line of text. Add the following code to test the function:

```
csv = loadStrings('playlist.csv')

for entry in csv:
    print(entry)
```

The *playlist.csv* data is assigned to a list named csv. Each csv element holds a line of text representing a single track. The for loop prints each entry on a new line in the console:

```
location,title,creator,album,trackNum
file:///music/SpeakToMe.mp3,Speak to Me,Pink Floyd,The Dark Side of the Moon,1
file:///music/Breathe.mp3,Breathe,Pink Floyd,The Dark Side of the Moon,2
. . .
```

The `loadStrings()` function cannot distinguish between different plaintext formats; this could be a bestselling novel or the latest stock market figures.

To interpret the CSV data, use the `split()` method to break each line into further lists. In this case, you're splitting each entry so you can extract the number and title of each track; you don't need the file location, creator, or album. The `split()` method works by using a delimiter argument of your preference. In this case, you'll use a comma. Amend your `for` loop code like this:

```
...
❶ for entry in csv[1:]:
    ❷ track = entry.split(',')
    print('{}. {}'.format(track[4], track[1]))
```

By adding `[1:]`, the `for` loop skips the first item in the `csv` list ❶ to avoid printing the field headings. With each iteration, the `split()` method assigns a new list to the `track` variable ❷. The elements `tracks[4]` and `track[1]` hold the entry track number and title, respectively.

Run the sketch to confirm that the console displays a list of 10 numbered tracks:

```
1. Speak to Me
2. Breathe
...
```

If you want to write text to a file, look up the `saveStrings()` function in the online Processing reference; it's effectively an inverse `loadStrings()`.

Formatting plaintext data in CSV files is a good way to avoid having to manage your data in the Processing editor. The beauty of CSV lies in its simplicity, but it isn't great for dealing with hierarchically structured data. In Chapter 8, you'll learn about other text-based formats (XML and JSON).

NOTE *Python provides a* csv *module to deal with CSV data, and it's worth exploring if you want to do more advanced CSV processing.*

Challenge #8: Games Sales Chart

In this final challenge, you'll generate a bar chart of the bestselling video games of all time. Figure 7-15 presents the final result (left) along with a zoomed-in version to provide more detail (right).

Figure 7-15: Completed chart (left) and chart detail (right)

The data has been sourced from a Wikipedia article titled "List of best-selling video games" (*https://en.wikipedia.org/wiki/List_of_best-selling_video_games*) and converted from an HTML table to a tab-separated file. The rankings likely have shuffled since this book was published, but that doesn't matter for the purpose of this exercise.

You'll read in the sales data by using a loadStrings() function, and then plot the chart by using the techniques you've learned in this chapter. Create a new sketch named *game_sales_chart* with a *data* subfolder. In the preceding exercise, you downloaded a *data.zip* file, which also contains a *list_of_best-selling_video_games.tsv* file; place this in the sketch *data* subfolder.

This file uses tab-separated values, hence the *.tsv* file extension. I used tabs because it's highly unlikely that any game titles or studio/publisher names will contain tab characters, but there may be commas that could interfere with a split(',') style approach. You may want to open the TSV file in your preferred spreadsheet application to inspect the values. There are 50 games in all, ordered with the bestselling game at the top. If you use a text editor to open the file, you should see something like this:

```
Rank    Title   Sales   Developer(s)    Publisher(s)
1       Minecraft       180000000       Mojang  Xbox Game Studios
2       Tetris  170000000       Elektronorgtechnica     Various
. . .
```

A single, invisible tab character separates each field. Note that tab sizes may vary among editors and will not always form visually aligned columns, so the file may look a little different, depending on the editor you use.

Add basic setup code to your sketch that will define the display window size and background color, as well as read in the TSV data:

```
size(800, 800)
background('#004477')
tsv = loadStrings('list_of_best-selling_video_games.tsv')
noStroke()
```

A list named `tsv` holds the game sales entries. None of the graphic elements have strokes, so I've included a `noStroke()` line.

You'll need to perform calculations to scale the bars relative to the display window. Although the sales figures appear to be numbers, Processing treats them as text. Recall that you cannot perform mathematical operations on string data. Fortunately, there's an easy fix. The `int()` and `float()` functions convert various data types to integer and floating-point values, respectively. Here's an example:

```
entry1 = tsv[1].split('\t')  # Minecraft entry
sales1 = entry1[2]           # 180000000
print(int(sales1) + 1)       # 180000001
```

The `split()` method must create a list from the first entry (Minecraft) using a tab character as a delimiter; to specify a tab, use `'\t'` as an argument. The variable `sales1` is equal to the value at index 2, the Sales column. Despite looking like a number, this value is a string, so the `print` line wraps `sales1` with an `int()` function to convert it to an integer before adding 1 to it.

Now, complete the chart as shown in Figure 7-15. It's probably best to start with a loop that prints each entry. Then, get the labels to display before creating the bars. Once you have the labels, create plain white bars of the correct width, and finish it off with the rainbow sequence effect. If you need help, you can access the solution at *https://github.com/tabreturn/processing.py-book/tree/master/chapter-07-working_with_lists_and_reading_data/game_sales_chart/*.

Summary

In this chapter, you learned about Python's suite of methods for various list operations, how to manage collections of items using lists, and how lists are particularly powerful when combined with loops. You also learned to harness nested lists in order to manage more complex data and practiced a few data visualization techniques.

In addition, you saw how to work with data stored in plaintext formats, like CSV and TSV, allowing you to read in values from external files when you run a sketch. This means you don't need to manage values in the Processing editor, making it easier to swap out datasets.

The next chapter moves on to dictionaries, which are similar to lists in that they store collections of items. With dictionaries, however, you access values by using a word instead of an index. Once again, you'll create novel data visualizations with your new skills.

8

DICTIONARIES AND JSON
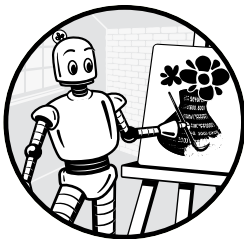
Dictionaries hold collections of items, similar to the ordered lists you learned about in Chapter 7. Dictionaries, however, are unordered, and you use an associated value to access each item, which makes it easier to remember what the items in your dictionary represent. In this chapter, you'll learn about Python's dictionary syntax and methods, how to combine loops and dictionaries, and how to nest dictionaries and lists.

You'll also learn how to work with another plaintext file format: *JavaScript Object Notation (JSON)*. The syntax isn't quite as simple as CSV, but it's better suited for handling more complex data structures. You'll use Python's built-in *json* module to read in dictionary data from a JSON file, and as in Chapter 7, you'll then create a data visualization.

Introducing Dictionaries

In a dictionary, each unordered item is associated with a value called a *key*. A key is usually a short string, and each dictionary item is composed of a key-value pair. This means dictionaries are associative, and some programming languages refer to dictionary-type structures as *associative arrays*. Dictionaries are different from lists, which are numerically indexed, because each element in a list corresponds to a number (index) indicating its position in a sequence of items.

As an example of how key-value pairs work, you might use a dictionary to note the favorite movie of each of your friends. In such a dictionary, each key would be a friend's name, and each value would be a corresponding film title. To insert or retrieve your friend Lee's favorite movie, you'd use the key 'Lee'. Figure 8-1 is a conceptual diagram of this dictionary.
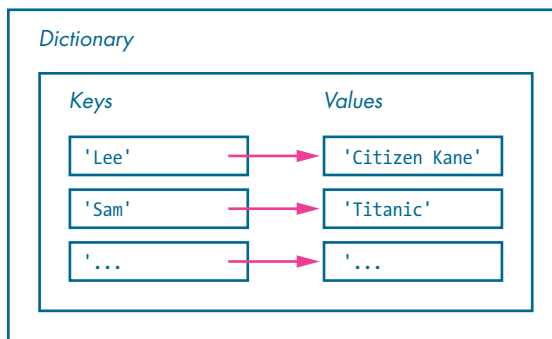
Figure 8-1: A diagram of a dictionary indicating the mapping between keys and values

For your first exercise in working with Python dictionaries, you'll write code to manage student records. Create a new sketch named *dictionaries*, and add the following code that shows the difference between a list and dictionary:

```
❶ student = ['Sam', 24]
❷ student = {'name': 'Sam', 'age': 24}
```

First, note that dictionaries use curly brackets ({}), whereas lists use square brackets ([]). The list ❶ and dictionary ❷ variants store the same values: 'Sam' and 24. However, each dictionary item includes a value *and* a key. In this instance, the dictionary keys are 'name' and 'age'. Keep in mind that sensibly named keys help identify what the values represent. This student dictionary holds two key-value pairs (Figure 8-2).

You'll use the meaningfully named keys to retrieve values from the student dictionary. To use the student list, you would need to recall the seemingly arbitrary positions for each value. Lists are better at handling an ordered sequence of items, but if you have a set of unique keys that map to values, use a dictionary.

```
student = { 'name':'Sam', 'age':24 }
                  |        |        |     |
                 Key     Value     Key   Value
              └──────────────┘   └──────────────┘
                Key-value pair     Key-value pair
```
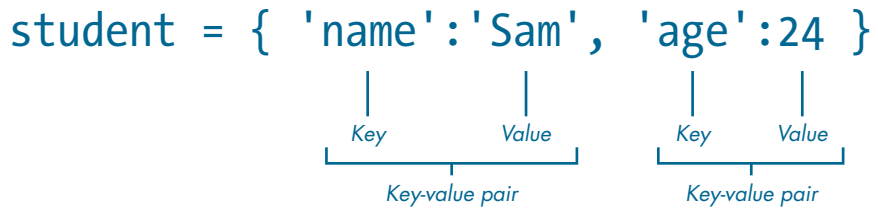
Figure 8-2: A dictionary with two key-value pairs

Dictionaries can hold all types of data, including strings, numbers, Booleans, lists, and even other dictionaries. You can store as many key-value pairs as you like in a dictionary. Technically speaking, there is an upper limit, but if you're managing such large volumes of data, you should probably look at a database solution.

Accessing Dictionaries

To access any dictionary value, use the dictionary name along with the associated key in square brackets. Try the following code:

```
. . .
print(student['age'])   # displays: 24
print(student['name'])  # displays: Sam
```

The comments alongside each `print()` function confirm what should appear in your console.

To print the entire dictionary, omit the square brackets and key, leaving only the dictionary name:

```
print(student)          # {'name': 'Sam', 'age': 24}
```

The console should display every key-value pair, complete with the curly brackets, colons, and commas.

The order of the key-value pairs won't always match the order in which you defined them, and this can vary among Python environments. Dictionaries are inherently orderless; Python is concerned with the connections between the keys and values. If you need to order your dictionary items—by key or value—you can use various functions and methods. You'll see some sorting techniques in "Combining Loops and Dictionaries" on page 163.

If you try to reference a nonexistent key—say, `student['grade']`—Processing displays a `KeyError` message. If you need to check whether a key exists, use the `in` operator:

```
. . .
if 'age' in student:
    print(student['age'])
```

The `in` checks whether the `'age'` key exists in the `student` dictionary. If found, the operation returns `True`, and the `if` statement executes the `print` line displaying the age value (which is `24` in this case).

Modifying Dictionaries

Dictionaries are dynamic structures, so you can add and modify key-value pairs as you please. To change an existing value, reassign it as you would a list element, but use the key as opposed to a numeric index. To illustrate, in your *dictionaries* sketch, change the student's age to **25**:

```
. . .
student['age'] = 25
print(student)          # {'name': 'Sam', 'age': 25}
```

The console output should confirm that the age has changed from `24` to `25`.

To add a new key-value pair, follow the same process. Add a student ID number to the `student` dictionary:

```
. . .
student['id'] = 19950501❶
print(student)          # {'name': 'Sam', 'id': 19950501, 'age': 25}
```

The `id` value here represents a birth date (1995-05-01) ❶. The system can use this information to calculate the student's age, so now it's no longer necessary to store the individual's age value. To remove this, use the `del` statement:

```
. . .
del student['age']
print(student)          # {'name': 'Sam', 'id': 19950501}
```

The `del` statement permanently removes the `age` key and its corresponding value.

Nesting Dictionaries and Lists

Dictionaries can hold other dictionaries or lists, and lists also can hold dictionaries. Let's look at two examples: a dictionary of lists, and a list of dictionaries. Both are valid ways to structure data, but as you'll see, you'll choose one depending on which works best in your particular application.

At this point, your program stores the details of a single student. A system that can manage just one student isn't very useful, so to handle multiple students, try a *dictionary of lists*. Add the following code to the bottom of your *dictionaries* sketch:

```
. . .
students = {
❶ 'names': ['Sam', 'Lee'],
```

```
❷  'ids': [19950501, 19991114]
}
print(students['names'][1]❸)  # Lee
```

The `names` list item holds a list of two names ❶; the `ids` list holds their respective ID numbers ❷. To access any list element directly within a dictionary item, use the associated key followed by a second pair of square brackets containing the element index ❸.

Another approach to structuring this data is with a *list of dictionaries*. Rather than separating the student names into one list and the IDs into another, you could use a dictionary for each student. Add the following code to the bottom of your sketch, effectively overwriting the former `students` dictionary entirely:

```
. . .
students = [
  {'name': 'Sam', 'id': 19950501},
  {'name': 'Lee', 'id': 19991114}
]
print(students[1]['name']❶)   # Lee
```

This retrieves the name `Lee` by using an element index, followed by another pair of square brackets containing the dictionary key ❶.

The latter of the two approaches (the list of dictionaries) is arguably the more sensible structure in this scenario. Each item is like a row in a spreadsheet that contains the details for a single student, and each student may have a different number of columns. This means you can add an extra key-value pair to Sam's dictionary without having to do the same for Lee. Doing this, however, would be tricky using the first approach.

What you name your keys and how you elect to nest lists and dictionaries should help relate your data to real-world models while reducing complexity. Keep your key names short and descriptive, and bear in mind that well-structured data will make for more self-evident algorithms further along in your program. In other words, you'll end up saving time and energy writing code if it's structured around intuitively organized data.

Combining Loops and Dictionaries

You'll often want to loop through your dictionaries. For example, you could dynamically generate reports for every student in your system by using a single loop to populate a predefined template. Considering that a dictionary can hold thousands or even millions of key-value pairs, this is a powerful technique. Because of the key-value system, though, iterating dictionaries is a little different from iterating lists.

You can iterate a dictionary's keys, values, or key-value pairs with Python's `keys()`, `values()`, and `items()` methods. Note that many list methods—such as `append()`, `extend()`, `index()`, `insert()`, and `remove()`—do not work on the dictionary data type.

To begin, add a new dictionary, named `courses`, to the end of your *dictionaries* sketch:

```
. . .
courses = {
    'game development': 'Prof. Smith',
    'web design': 'Prof. Ncube',
    'code art': 'Prof. Sato'
}
```

The dictionary keys represent course titles; the associated values are the professors who coordinate each course. Next, you'll look at ways to combine this dictionary with `for` loops.

Iterating Keys

You can write a `for` loop that deals only with keys, which is useful if you don't need to work with the values in your dictionary. Key iteration happens implicitly wherever you use a `for...in` line with a dictionary. Test this behavior in the following example, which should display all of the course titles in your console:

```
. . .
for course in courses:
    print(course)
```

With each iteration, Python assigns the next key in `courses` to `course`. The `print` line displays each course title on its own line in the console, and the loop is complete after all of the keys are exhausted:

```
web design
game development
code art
```

Recall that you cannot rely on the ordering of dictionary items. If you want to ensure that keys are retrieved in alphanumeric order, wrap a `sorted()` function around `courses`:

```
. . .
for course in sorted(courses):
    print(course)
```

The amended `for` line prints the keys in the following order:

```
code art
game development
web design
```

If you simply need a list of keys, use the keys() method, and to sort them, include a sorted() function:

```
print(sorted(courses.keys()))
# displays: ['code art', 'game development', 'web design']
```

This prints a list of the keys in the console, complete with square brackets and commas.

Iterating Values

The values() method returns all of a dictionary's values, which is useful if you don't need to work with the keys in your dictionary. Add a new loop to your sketch, using the values() method to retrieve the names of each professor:

```
. . .
for prof in courses.values():
    print(prof)
```

With each iteration, Python assigns the next value in courses to prof. I've named this variable prof as an abbreviation for *professor* (representing the value it will hold). The print line displays the name of each course's professor on a new line in the console.

Iterating Items

Often you'll want both the dictionary key and the corresponding value for your loop. The items() method returns all of the dictionary's key-value pairs. Before writing any loop code, print the items in your courses dictionary by using the items() method:

```
print(courses.items())
```

Here's the console output (it doesn't fit on a single line, which is indicated with the ellipsis):

```
[('web design', 'Prof. Ncube'), ('game development', ...
```

You should be able to identify each key-value pair, grouped within parentheses. The parentheses surrounding each key-value pair denote a *tuple*. This book does not cover tuples in detail, so for now, consider them interchangeable with lists.

Add this loop to print each key-value tuple on its own line in the console:

```
. . .
for kv in courses.items():
    print(kv)
```

You can pronounce tuple as *too-ple* or *tuh-ple*, depending on whom you want to annoy.

To access tuple items, use the same syntax as lists. Here's an example:

```
newcourse = ('visual effects', 'Prof. Kovalenko')
print(newcourse[0])  # visual effects
print(newcourse[1])  # Prof. Kovalenko
```

In the first line, a new tuple, defined using parentheses, is assigned to a variable named newcourse. Tuple elements are numerically indexed, relying on the list-style syntax to retrieve values, as you can see from the print lines. In a nutshell, the key difference between lists and tuples is that tuples, once defined, cannot be modified. The technical term for this quality is *immutable*. For example, the following code would result in a TypeError because you're attempting to alter a tuple value:

```
newcourse[0] = 'VFX'
```

Conversely, lists and dictionaries are *mutable* data types, because you can add, remove, or edit items as you please.

When you use an items() method with a for loop, Python is assigning a tuple to your loop variable. I've named this kv as an abbreviation for *key-value*, but you could name it whatever you like. For example, on the *code art* iteration, kv is equal to ('code art', 'Prof. Sato'). Run the sketch to confirm that the console displays each key-value pair, complete with parentheses and commas.

To make iterating dictionary items more convenient, Python allows you to include two variables between the for and in, one for the key and one for the corresponding value. You can name these variables whatever you like, but the order of assignment is always *key* first, *value* second; this matches the ordering in the tuple. Add this example to assign the course title and professor to separate variables. Additionally, this code includes a sorted() function:

```
. . .
for course❶, prof❷ in sorted(courses.items()):
    print('{} coordinates the {} course.'.format(prof, course))
```

With each iteration, Python assigns the key (course title) to the first variable ❶ and the value (professor name) to the second variable ❷. The console should display the following:

```
Prof. Sato coordinates the code art course.
Prof. Smith coordinates the game development course.
Prof. Ncube coordinates the web design course.
```

Notice that the `sorted()` function always operates on keys, so the sentences are ordered alphabetically by course title, not professor. To reverse the order, add a `reversed()` function:

```
. . .
#for course, prof in sorted(courses.items()):
for course, prof in reversed(sorted(courses.items())):
    print('{} coordinates the {} course.'.format(prof, course))
```

Now, the `code art` course will be listed as the last line in your console.

Working with JSON

JavaScript Object Notation was derived from JavaScript, but it's a language-independent data format. Many programming languages support JSON, Python included, and it's popular for web development. You can use JSON to store dictionary-like data in plaintext files, with key-value pairs, to construct nested dictionary- and list-style structures.

For this exercise, you'll use JSON to format data in a plaintext file, just as you did with CSV in Chapter 7, except with a different syntax and the *.json* file extension. Python's built-in *json* module will handle the data you read in. As mentioned previously, JSON syntax isn't quite as simple as CSV, but it's more descriptive and versatile.

LIBRARIES AND MODULES

A *module* is a set of prewritten, reusable code. You can write your own modules or import an existing module that contains the functionality you require. A *library* is a collection of modules. You'll use a few libraries and modules in the chapters to come.

Understanding JSON Syntax

To understand how JSON syntax works, let's contrast it with CSV. In Chapter 7, you stored an album track list (for *The Dark Side of the Moon*) in CSV format. Here's an abridged version of that file:

```
location,title,creator,album,trackNum
file:///music/SpeakToMe.mp3,Speak to Me,Pink Floyd,The Dark Side of the Moon,1
file:///music/Breathe.mp3,Breathe,Pink Floyd,The Dark Side of the Moon,2
. . .
```

The first line contains the field headings. Lines 2 and beyond provide the details of each track.

Here's the same (abridged) track list formatted as JSON:

```
[
  {
    "location": "file:///music/SpeakToMe.mp3",
    "title": "Speak to Me",
    "creator": "Pink Floyd",
    "album": "The Dark Side of the Moon",
    "trackNum": 1
  },
  {
    "location": "file:///music/Breathe.mp3",
    "title": "Breathe",
    "creator": "Pink Floyd",
    "album": "The Dark Side of the Moon",
    "trackNum": 2
  },
  . . .
```

Each value has a corresponding key—just like a Python dictionary! If you study the code, you'll realize that it looks like a list of dictionaries written in Python. However, subtle differences exist between JSON and Python's data structure syntax. For instance, with JSON, you must use double quotes for strings; in Python, you have the option of single quotes. The terminology is a little different too. In JSON, curly brackets denote an object (not a dictionary), and square brackets define an array (not a list). What you name your keys and how you nest your elements is up to you.

N O T E   *Python permits an optional trailing comma after the last element in any list or dictionary. JSON, however, disallows trailing commas in objects and arrays.*

Because this is a track list for an individual album, every track has the same `creator` and `album` information, which seems redundant. To avoid repetition, you can restructure your JSON as follows:

```
{
  "creator": "Pink Floyd",
  "album": "The Dark Side of the Moon",
```

```
    "tracklist": [
      {
        "location": "file:///music/SpeakToMe.mp3",
        "title": "Speak to Me",
        "trackNum": 1
      },
      {
        "location": "file:///music/Breathe.mp3",
        "title": "Breathe",
        "trackNum": 2
      },
      . . .
```

The new structure nests the tracks within `tracklist`. The `creator` and `album` information is placed at the top level of the structure because it applies to every track.

You can write your own JSON data, generate it dynamically, or source it online.

Using Web APIs

Vast repositories of JSON data, ranging from music metadata to cat facts, are available via web APIs. A *web application programming interface (API)* is a web-based service you can use to request or post data. For example, you might request data from the Twitter API to generate a graph measuring your tweet frequency or to program a Twitter bot that autonomously posts tweets of your code art.

This book doesn't cover how to use web APIs. However, if you want to explore web APIs, you should know a couple of things. Each API works slightly differently, which means you'll need to refer to the service-specific developer documentation. Many APIs are accessible directly via a URL, which allows you to interface with them by using your web browser. For example, *OpenAQ* provides air-quality data from around the world. If you enter the following URL into your web browser, you'll get a JSON summary of the air-quality data for every city in Norway: *https://api.openaq.org/v1/cities?country=NO*.

The *api.openaq.org* part is the API domain name. The */v1* indicates that you are using version 1, the first release of the API. The */cities* part requests the data for every city in OpenAQ's database, but the *?country=NO* limits the cities to those located in Norway. For a copy of the data, use the **Save Page As** option in your browser menu, or copy and paste the contents into any plain-text editor.

You're also likely to come across APIs that provide CSV and XML data.

CSV, JSON, and XML have their own strengths and weaknesses, so weigh the relative merits of each format when considering what's best for your projects. The beauty of CSV lies in its simplicity, but it cannot support hierarchically structured data. Unlike JSON, which allows you to nest objects within objects several levels deep, CSV limits you to a single value per field. XML is an established, widely supported, and flexible data exchange format,

but it can be overly complex and bloated at times. JSON provides somewhat of a middle ground, and it has become increasingly popular on the web, as its syntax is more concise than XML's.

XML

Extensible Markup Language (XML) files are plaintext files with the *.xml* extension. To give you an idea of what the syntax looks like, here's an XML adaption of *The Dark Side of the Moon's* track list:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tracklist creator="Pink Floyd" album="The Dark Side of the Moon">
  <track>
    <location>file:///music/SpeakToMe.mp3</location>
    <title>Speak to Me</title>
    <trackNum>1</trackNum>
  </track>
  <track>
    <location>file:///music/Breathe.mp3</location>
    <title>Breathe</title>
    <trackNum>2</trackNum>
  </track>

  . . .
```

Even if you have never written or viewed any XML before, you can likely make some sense of the playlist contents in the code. You can discern the details of each track within opening and closing pairs of track tags (`<track>...</track>`). The opening `tracklist` tag includes two attributes that contain the `creator` and `album` information.

You're likely to encounter XML, or a similar markup language (like HyperText Markup Language, or HTML), on your programming journeys. Python provides various modules for dealing with markup languages, but I do not cover them in this book.

Reading in JSON Data

You can read in JSON data when your sketch runs. In this example, you'll use coffee data stored in a JSON file to generate a chart. Create a new sketch named *coffee_chart* with a *data* subfolder, and then complete the following steps:

1. Open your browser and go to *https://github.com/tabreturn/processing.py-book/*.
2. Navigate to *chapter-08-dictionaries_and_json*.
3. Download the *data.zip* file.
4. Extract the ZIP archive, and move *coffees.json* to the sketch *data* subfolder.

Here is a snippet of the *coffees.json* file contents:

```
[
  {
    "name": "Espresso",
    "ingredients": [
      {"ingredient":"espresso"❶, "quantity":30 ❷, "color":"#221100"❸}
    ]
  },
  . . .
  {
    "name": "Irish Coffee",
    "ingredients": [
      {"ingredient":"espresso", "quantity":60, "color":"#221100"},
      {"ingredient":"whiskey", "quantity":40, "color":"#FFCC77"},
      {"ingredient":"whippedcream", "quantity":20, "color":"#FFFFFF"}
    ]
  },
]
```

Each top-level object holds the details for a different type of coffee. This code shows two types of coffee: `Espresso` and `Irish Coffee`. These are the first and last recipes; there are nine types of coffee in all. Each `ingredient` object has three key-value pairs: an ingredient name ❶, quantity in milliliters ❷, and fill color ❸. Note that these quantities are not necessarily accurate, so the final chart might not impress baristas and coffee aficionados, but it will look pretty cool.

The next step is to load the data from the *coffees.json* file. Python's `open()` function can open any file, plaintext or otherwise, and return a file object. For a JSON file, load the file object into a Python data structure by using the built-in *json* module. Add this code to your sketch:

```
import json
jsondata = open('coffees.json')
coffees = json.load(jsondata)
```

The `import` line imports the *json* module. The `open()` function opens the JSON file and assigns the file object to the variable `jsondata`. The `json.load()` function converts the JSON to Python data. To confirm it's working, print the quantity of whiskey in the Irish Coffee:

```
print(coffees[8]['ingredients'][1]['quantity'])  # 40
```

The Irish Coffee is the last element in a list of nine coffees; hence, it has an index of 8. The whiskey content is the second ingredient (`['ingredients'][1]`). The final `['quantity']` represents the quantity value of the ingredient, 40 milliliters.

When you retrieved CSV data by using `loadStrings()`, everything was data-typed as a string, numbers included. You had to convert values to integers by using the `int()` function before performing any arithmetic; to create lists, you had to use a `split()` function. The *json* module, however, handles all of this

conversion for you. A JSON value like 40, with no decimal point or quotation marks, is interpreted as an integer; comma-separated values within square brackets are turned into Python lists automatically; and so forth. Now that you can access the data in Python, you can use it to render a chart.

Challenge #9: Coffee Chart

You'll visualize the data for all nine coffees by using nine mugs arranged in a 3 × 3 grid fashion. Figure 8-3 shows a screenshot of the final result.

Figure 8-3: The complete coffee chart

Add the following code to define a display window size, background color, and some variables, and to lay out nine empty mugs:

```
. . .
size(800, 800)
background('#004477')
mug = 120
spacing = 230
col = 1
```

```
translate(100, 100)

for coffee in coffees:
❶  # ingredients code goes here

    # mug
    strokeWeight(5)
    stroke('#FFFFFF')
    noFill()
    square(0, 0, mug)
    arc(mug, mug/2, 40, 40, -HALF_PI, HALF_PI)
    arc(mug, mug/2, 65, 65, -HALF_PI, HALF_PI)

    # label
    fill('#FFFFFF')
    textSize(16)
    label = coffee['name']
    text(label, mug/2-textWidth(label)/2, mug+40)

❷  if col = 3:
        translate(spacing*-2, spacing)
        col = 1
❸  else:
        translate(spacing, 0)
        col += 1
```

The lines above the for line define the main parameters for the sketch, such as the display window size and background color. The mug and spacing variables control the mug sizing and spacing, respectively; the col variable serves as a column counter. Comments within the for loop indicate where the mug and label drawing code begins. After drawing each mug ❸, Processing translates the drawing space 230 pixels to the right and adds 1 to col. When col reaches 3 ❷ (every third mug), the drawing space is shifted back to the left edge of the display window and down one row, and col resets to 1. The ingredients code has been left for you to complete; the comment indicates where you should write it ❶.

Run the sketch. You should see nine empty mugs with labels. Now, complete the chart so it looks like Figure 8-3. If you need help, you can access the solution at *https://github.com/tabreturn/processing.py-book/tree/master/chapter-08-dictionaries_and_json/coffee_chart/.*

Summary

In this chapter, you learned to organize a collection of items in a dictionary that can associate values with meaningfully named keys. Furthermore, you combined dictionaries and lists to create more intuitive data structures. You also learned how to define, access, modify, and nest dictionaries, and how to loop through dictionaries by using keys, then values.
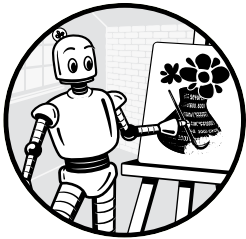
This chapter also introduced you to JSON. You learned how it's similar to Python dictionaries and lists, and how to read in JSON data. You can store

dictionary and list data in JSON files to separate your Python code from your data. If you're looking for interesting data to work with, many online sources host JSON datasets that you can access for free.

In Chapter 9, you'll learn how to define and work with functions, which are named sections of code. You decide what to name your functions, and whenever you want to run a function, you call it by its name. This helps reduce repetition in your code because you can repeat a one-line function call instead of many lines of code. Think of functions as reusable blocks of code that will make your sketches more efficient and easier to maintain. You'll write functions, including some to generate elliptical and wave-like motion, and then you'll use these functions to program animated effects that employ trigonometry.

9

FUNCTIONS AND PERIODIC MOTION

As your programs grow more complex, your line counts will increase, and you'll begin repeating the same or similar code. By using *functions*, you can divide your programs into named blocks of reusable code. This makes your code more modular, allowing you to reuse lines without needing to rewrite them.

You've already used many Processing functions, like `size()`, `print()`, and `rect()`, and in this chapter, you'll learn how to define your own functions. As an example, Processing has no function for drawing diamonds, but you can create one. You decide what to name this function and what arguments it will accept. Perhaps your `diamond()` function accepts an x, y, width, height, and optional rotation argument.

You'll also create functions for generating elliptical and wave-type motion, which will involve delving into some trigonometry. You'll incorporate the mathematical functions sine and cosine by using Processing's built-in functions for performing these calculations. If the mention of

trigonometry triggers disturbing flashbacks from math class, take a deep breath and relax. This will be a practical and visual reintroduction to these concepts, with Processing crunching all the numbers for you.

Defining Functions

Sensibly named functions make your code easier to understand and work with. A 1,000-line program can be tricky to comprehend, especially for somebody who didn't write it.

Imagine programming a music player. You might create a function named `play()` that executes 20 or so lines of code necessary to load and play an MP3 file. When you need to play a track, you simply call your `play()` function by using a file argument, like `play('track_1.mp3')`. You don't need to concern yourself with the details of how the `play()` function operates after you've defined it, and neither does anybody else working with your code. Additionally, you could define functions for `stop()`, `pause()`, `skipBack()`, and `skipForward()`.

In this section, you'll learn to define functions with the `def` keyword and then how to handle arguments. You might call these *user-defined functions* to distinguish them from those that come built-in with Python and Processing.

Creating a Simple Speech Bubble Function

Let's begin with a simple function that takes no arguments and draws speech bubbles, like the ones you find in comic strips, in the console. You've already used functions that work without arguments, like Processing's `noFill()` that relies on just a function name and parentheses. Conversely, a function like `fill()` requires at least one argument, such as a hexadecimal color value.

Your speech bubble function will form an outline, using plaintext characters, that surrounds a caption. Once you have this working, you'll move on to defining a more dynamic function that accepts a range of arguments to draw speech bubbles in the display window.

Create a new sketch and save it as *speech_bubbles*. Add the following code that prints a question in the console, followed by the answer in a speech bubble five seconds later:

```
  wait = 5000
❶ print('1. What do you get if you multiply six by seven?')
❷ delay(wait)
  print(' ------------------ ')
  print('| The answer is 42! |')
  print('| ---------------- ')
  print('|/')
```

When you run the sketch, you should see the question appear in your console immediately ❶. The Processing delay() function halts the program for 5,000 milliseconds (five seconds) ❷, then reveals the answer in a speech bubble using the four print lines that follow it. Run the sketch to confirm this:

```
1. What do you get if you multiply six by seven?
 ------------------
| The answer is 42! |
| ------------------
|/
```

This might not look like the most convincing speech bubble, but it'll do for now.

Make the following changes to your code to define a function for printing the answer:

```
wait = 5000

def printAnswer():
    print(' ------------------ ')
    print('| The answer is 42! |')
    print('| ----------------- ')
    print('|/')

print('What do you get if you multiply six by seven?')
delay(wait)
printAnswer()
```

The def keyword defines a new function. You can name this function whatever you like, but make the name descriptive. Like variable names, function names should contain only alphanumeric and underscore characters, and they must start with a letter or an underscore; in this case, I've chosen printAnswer. Always include the parentheses and a colon at the end of the def line. The four print() lines are in the *body* of the function definition, which is the indented section of code beneath the def line. The function won't execute the print lines until you *call* it. On the last line, where the program must reveal the answer, is the printAnswer() function call.

NOTE *Python will process your code line by line, beginning at the top of the file. If it attempts to execute a function call before it has processed the corresponding definition, the program will fail. In other words, you cannot call printAnswer() on the first line of your sketch, because Python would not yet have encountered the def printAnswer() line.*

When you run the sketch, the program should work as before, printing the question followed by the answer in a speech bubble five seconds later.

The PEP 8 style guide recommends that "function names should be lowercase, with words separated by underscores as necessary to improve readability." In other words, the printAnswer() function instead should be named print_answer(). However, when an existing style is established, internal consistency is usually preferred.

   I've opted for a camelCase function name to match the convention used for Processing's built-in functions, like noFill() or pushMatrix(). As noted in Chapter 1, camelCase combines multiple words into one and uses a capital letter to start the second and subsequent words. The style is also referred to as *mixedCase*, or sometimes *lowerCamelCase* (to contrast it with *UpperCamelCase*).

   Add a second question to the end of your sketch:

```
. . .
delay(wait/2)
print('2. How many US gallons are there in a barrel of oil?')
delay(wait)
printAnswer()
```

After displaying the answer to question 1, the program waits two and a half seconds and prints question 2. The answer to question 2 is revealed five seconds after this. Once again, the answer is 42, but there's no need to retype the four lines of code for displaying the speech bubble. Instead, you can call the printAnswer() function a second time.

   You can add as many questions as you like. If the answer to each question is 42, you can call the printAnswer() function to display the answer. If you want to restyle *all* of your speech bubbles—for example, using different characters for the outline—edit the body of the printAnswer() definition. You need to change the code in only one place to affect every speech bubble.

For each answer, you have a neat, one-line function call with a name that indicates what it does. Other programmers won't need to understand the inner workings of the printAnswer() function to use it, but if necessary, they can read through the definition code to find out how it works.

Before proceeding to the next section, set the wait value (at the top of your code) to 0:

```
wait = 0
. . .
```

This change cancels the effects of the delay() functions, because a delay time of zero means there is no delay. As a result, your sketch doesn't pause, and the next section of code you add can run immediately.

The printAnswer() function is limited to drawing speech bubbles in the console, and it always prints the same answer of 42, so next, you'll define a function that can accept arguments.

Drawing Compound Shapes Using a Function

To define a function that draws speech bubbles with shapes and text in the display window, continue working in your *speech_bubbles* sketch. First, you'll need an image over which to place your speech bubbles.

I've chosen Jan van Eyck's *Arnolfini Portrait* for this example because the painting has three speech bubble candidates: a man, a woman, and a dog. It's also public domain. Figure 9-1 presents the original painting on the left, and the result you're working toward on the right.

Figure 9-1: The original Arnolfini Portrait, 1434 (left); a version with speech bubbles (right)

You can download the *Arnolfini Portrait* image from Wikipedia (*https:// en.wikipedia.org/wiki/File:Van_Eyck_-_Arnolfini_Portrait.jpg*); the 561 × 768 pixel resolution will suffice. If you want to use a different image, that's fine too; just choose one with at least three subjects.

Create a new *data* subfolder and add your image to this; then add the following code to load and display it:

```
. . .
size(561, 768)
art = loadImage('561px-Van_Eyck_-_Arnolfini_Portrait.jpg')
image(art, 0, 0, width, height)
```

If you're not using the *Arnolfini Portrait*, adjust the size() and loadImage() arguments accordingly.

Run the sketch to confirm that the image spans your display window.

Define and then call a new speech bubble function by adding this code to the end of your sketch:

```
. . .
def speechBubble():
    x = 190
    y = 150
    txt = 'Check out my hat!'
    noStroke()
    pushMatrix()
    translate(x, y)

    # tail
    fill('#FFFFFF')
    beginShape()
    vertex(0, 0)  # tip
    vertex(15, -40)
    vertex(35, -40)
    endShape(CLOSE)

    # bubble
    textSize(15)
    by = -85
    bw = textWidth(txt)
    pad = 20
    rect(0, by, bw+pad*2, 45, 10)
    fill('#000000')
    textAlign(LEFT, CENTER)
    text(txt, pad, by+pad)

    popMatrix()

speechBubble()
```

If you're using a different image, adjust the x, y, and txt variables. The x and y variables control the position of the speech bubble—specifically, the x-y coordinate for the tip of the "tail" that's attached to the bubble. Before

drawing anything, a `translate()` function repositions the drawing space so that the vertex coordinates for this tip are (0, 0); the other tail vertices and the bubble are positioned relative to this point.

The `txt` variable defines the text that appears within the bubble. You can use any `txt` string you like, but keep it short. The speech bubbles will not accommodate multiline captions.

The code beneath the `bubble` comment draws a rounded rectangular bubble above the tail. The `rect()` function includes a fifth argument (`10`) that controls the corner radius. The larger you make this value, the rounder the corners become. The result is a rounded rectangular speech bubble with a tail at its bottom left (Figure 9-2).

Figure 9-2: The tip of the speech bubble tail has an x-y coordinate of (190, 150).

You can call the `speechBubble()` function 100 times, but the visual result always appears the same because every speech bubble draws over the one before it, at the same size, with the same text, in the same position. But, if you modify the x, y, and txt variables each time you call the `speechBubble()` function, you can customize the x-coordinate, y-coordinate, and caption. You can accomplish this by adding parameters to your function definition that allow you to pass values to the function using different arguments in your function call.

Adding Arguments and Parameters

Now you'll edit your `speechBubble()` definition so that the function can accept three arguments, allowing you to pass your coordinate and caption values to the function to manipulate the appearance of each speech bubble you draw. Arguments are assigned to corresponding *parameters*, but more on those shortly.

Currently, three variables control the speech bubble's appearance: x, y, and txt. To control those variable values via arguments, adapt your function definition as follows:

```
. . .
❶ def speechBubble(x, y, txt):
    #x = 190
```

```
#y = 150
#txt = 'Check out my hat!'
. . .
```

❷ `speechBubble(190, 150, 'Check out my hat!')`

The definition parentheses now include three parameters: x, y, and txt ❶. A parameter is a placeholder for a value that's provided by way of an argument. These parameters are made available within the local scope of the function; in other words, Python can access x, y, and txt only within the speechBubble() function block. You need to comment out (or delete) the old x, y, and txt lines to avoid overwriting the values that you pass in with the function call ❷.

Because you have three parameters, you must provide three arguments when you call the speechBubble() function. The first argument of 190 is assigned to parameter x, the second argument of 150 is assigned to parameter y, and so on, in the same order the parameters appear in the def line. These are called *positional arguments* because the order of the arguments determines which values are assigned to each parameter (Figure 9-3).

def speechBubble(x, y, txt):

(190, 150, 'Check out my hat!')
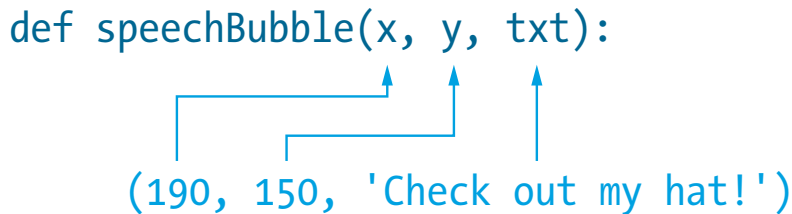
Figure 9-3: Positional arguments

Run the sketch to confirm that the visual result is unchanged. Try testing different arguments to change the appearance of the speech bubble.

NOTE   *It's not unusual to hear the terms* argument *and* parameter *used interchangeably. If you happen to mix them up, you aren't likely to confuse anybody.*

Call a second speechBubble() function:

```
. . .
speechBubble(315, 650, 'Woof')
```

The first and second (x and y) arguments position the speech bubble above the dog. The third argument specifies that the caption must read, "Woof" (Figure 9-4).

Figure 9-4: A second speech bubble

You now have a working `speechBubble()` function that accepts positional arguments. However, you can also call this function by using arguments in an arbitrary order if you use keyword arguments.

Using Keyword Arguments

When you call a function, you can state explicitly which value belongs to which parameter by using *keyword arguments*. These arguments include both a keyword and value. Each keyword takes its name from a parameter in the function definition. Consider this example, where both lines produce the same result:

```
speechBubble(315, 650, 'Woof')          # positional arguments
speechBubble(txt='Woof', x=315, y=650)  # keyword arguments
```

The first `speechBubble()` call employs a positional argument approach. The second call uses keyword arguments; notice that each value has a keyword in front of it. Python uses the keywords in your function call to match values and parameters (Figure 9-5).

Figure 9-5: Keyword arguments

This means you can order the arguments in your function call however you please. Just be sure to name your keywords exactly the same as the parameters in the function definition.

Setting Default Values

When you define a function, you can specify a *default value* for each parameter, which is like a backup Python can use if you leave out an argument in your function call. This behavior is useful for defining optional arguments. For example, the rect() function can accept an optional fifth argument for the corner radius. If you call the rect() function with four arguments, you get a rectangle with 90-degree corners, which is what users seem to want more often than not. But, if you provide the fifth argument (of something other than zero), you get a rectangle with rounded corners.

Use an equal sign to assign a default value to a parameter. For example, the following adds a default value of 'Hello' to your txt parameter:

```
. . .
def speechBubble(x, y, txt='Hello'):
    . . .
```

The default txt parameter is a string, but you can use any data type you like, including numbers and lists.

You can now call the speechBubble() function using two positional arguments, leaving txt (the third argument) to rely on its default value:

```
. . .
speechBubble(445, 125)
```

The 445 and 125 are positional arguments for x and y. As there's no third argument, txt defaults to 'Hello', as per the function definition. The result (Figure 9-6) is a speech bubble positioned above the woman's head that reads, "Hello."

Figure 9-6: Drawing a speech bubble using the default txt parameter, Hello

To replace *Hello* with *Meh*, call the speechBubble() function using three arguments:

```
. . .
speechBubble(445, 125, 'Meh')
```

Because you provided the positional argument for the `txt` parameter, the woman's speech bubble will now read, "Meh."

The lady clearly isn't overly impressed with her partner's hat, so she might choose not to risk offending him. A *thought bubble* could be more appropriate (Figure 9-7).

Figure 9-7: A speech bubble (left) and a thought bubble (right)

To draw a thought bubble, modify the `speechBubble()` function to draw a chain of small circles instead of a triangular tail. However, you want the `speechBubble()` function to depict speech bubbles by default, as they are more common than thought bubbles.

Add an additional `type` parameter to the function definition:

```
. . .
def speechBubble(x, y, txt='Hello', type='speech'):
    . . .
```

Now you have two parameters with default values. Notice that these come after the parameters with no default values. If you're defining any function with default values, place those parameters at the end of the list.

The next step is to modify the function body, specifically the section beneath the `tail` comment. The `type` parameter must determine whether Processing should draw a triangular tail or a chain of circles. Modify the code as follows:

```
    . . .
    # tail
    if type == 'speech':
        fill('#FFFFFF')
        beginShape()
        vertex(0, 0)  # tip
        vertex(15, -40)
        vertex(35, -40)
        endShape(CLOSE)

    elif type == 'thought':
        fill('#FFFFFF')
        circle(0, 0, 8)
```

```
    circle(10, -20, 20)
. . .
```

The `if` statement code will draw a triangular tail if the `type` parameter is equal to `'speech'`, the default value assigned in the function definition. The `elif` statement will draw a chain of two circles whenever the function call includes a `type` argument of `'thought'`. Edit your function call to see this in action:

```
. . .
speechBubble(445, 125, 'Meh', 'thought')
```

The `thought` argument switches the `speechBubble()` function to "thought bubble mode." If you omit this argument, the function defaults to drawing the speech bubble with the tail. Run the sketch to confirm that the result matches Figure 9-7.

Mixing Positional and Keyword Arguments

You can use positional arguments for your x and y coordinates, leave out the `txt` argument, and include a keyword argument for `type`. This way, Python can utilize the default value for `txt` (`'Hello'`), but render it in a thought bubble. As an example, you might want to replace the dog's speech bubble with a thought bubble that reads, "Hello." One option is to include a third argument of `'Hello'` explicitly in the function call—a fully positional approach. For example:

```
speechBubble(315, 650, 'Hello', 'thought')
```

Each argument here corresponds to a parameter. This seems redundant, though, given that `'Hello'` is the default value for parameter 3. If you just omit the `'Hello'` argument in your function call, Processing will draw a *speech* bubble with the word *thought* in it:

```
# a speech bubble that says, thought
speechBubble(315, 640, 'thought')
```

Recall that the third positional argument is for the `txt` parameter and that leaving out the fourth argument means Python has to adopt the default value for the fourth `type` parameter (speech bubble mode). A simple solution to this problem exists, however; use a keyword argument instead of relying on a positional argument:

```
speechBubble(315, 650, type='thought')
```

In this case, you've explicitly stated that the value `'thought'` belongs to the `type` parameter. You might notice that you can arrange the arguments in any order if you use keyword arguments for every value. This is true, so decide what combination of positional and keyword arguments works best in a particular situation.

If you're missing one or many required arguments in a function call, Processing displays an error message (Figure 9-8). For example, if you call the `speechBubble()` function with no arguments, the error message indicates that you require at least two.
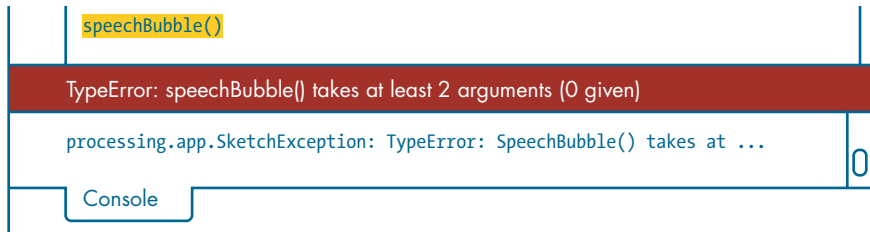
Figure 9-8: An error message for missing arguments

If you provide too many arguments, the error message indicates that `speechBubble()` takes at most four arguments.

Returning Values

You can use a function to operate on data and then have it *return* the result to the main program. This is different from the functions you've created so far, which execute a predefined section of code before resuming the regular flow of the main program.

To help explain this difference, here's some code to contrast a function that returns a value with one that does not:

```
x = random(100)
square(x, 40, 20)
```

Two Processing functions are in use here: `random()` and `square()`. The first one returns a value; the second does not. The `random()` function generates a floating-point value ranging from 0 up to but not including 100. The random function *returns* the value, which is assigned to a variable named x. The `square()` function draws a square in the display window; it does not return a value.

To define your own function that returns a value, use the `return` keyword. As an example, create a new function named `shout()`. This function accepts a single string argument, and then converts this string to uppercase and adds three exclamation marks to the end. Enter the following code above your `speechBubble()` calls to ensure that the `shout()` definition precedes any `shout()` function call:

```
. . .
def shout(txt):
    return txt.upper() + '!!!'
. . .
```

In the `return` line, the `upper()` method converts the string assigned to `txt` to uppercase; the final result is a concatenation of this and three exclamation

marks. Once Python processes the `return` statement, it exits the function immediately. In other words, if you add any further code to the `shout()` definition below the `return` line, Python ignores it.

You could use this function to add emphasis to the text in any speech bubble. Here's an example:

```
speechBubble(190, 150, shout('Check out my hat'))
```

The `shout()` function converts the string to "CHECK OUT MY HAT!!!" before it's passed to the `speechBubble()` function. This wraps the argument with `shout()` to avoid having to create an intermediate variable, which you would then pass to the `speechBubble()` function.

This was a simple example to introduce how the `return` keyword works. Many functions that return values perform more complex tasks, like Processing's `sqrt()` function that calculates the square root of any number.

Defining Functions for Periodic Motion

In this section, you'll learn how to simulate periodic motion in Processing by defining functions that employ trigonometry to draw circular patterns and waves. In physics, *periodic motion* is motion that repeats itself at regular intervals, such as a swinging pendulum, waves moving through water, or the moon orbiting the Earth. A *cycle* is one complete repetition of the motion. The *period* is the time it takes to complete a cycle. The period for the moon's orbit of the Earth is roughly 27.3 days; the second hand of a clock has a period of 60 seconds.

Trigonometry, or *trig*, is a branch of mathematics that studies triangles and uses various mathematical functions, such as sine and cosine, to calculate angles and distances. It also has applications in many fields of programming. For instance, games that incorporate physics must continuously calculate the position and speed of objects in motion, and those calculations involve triangles.

Trig is also useful for controlling steering and aiming behavior. For example, if you know the x-y coordinates of the player and enemy turret in Figure 9-9, you can calculate how to rotate the enemy gun to aim it at the player.

You'll use right triangles to calculate points along the circumference of a circle, using sine and cosine functions. The coordinates for those points are what you use to simulate smooth, periodic motion.
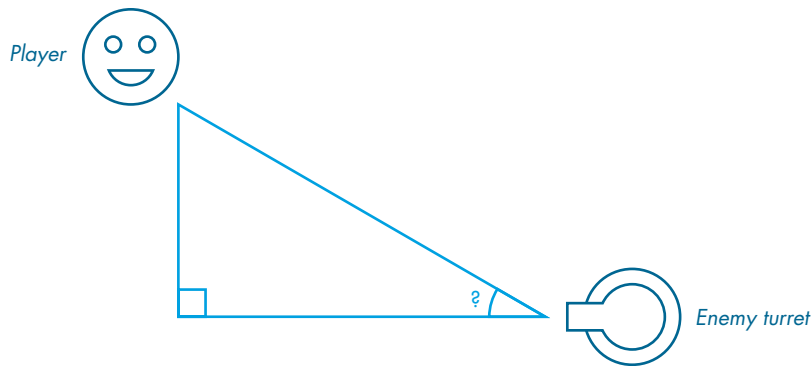
Figure 9-9: If only the enemy turret had listened in math class.

Create a new sketch and save it as *periodic_motion*. Add the following code to set up the drawing space:

```
def setup():
    size(800, 600)

def draw():
    background('#004477')
    noFill()
    strokeWeight(3)
    stroke('#0099FF')
    line(width/2, height, width/2, 0)
    line(0, height/2, width, height/2)
    # flip the y-axis
    scale(1, -1)
    translate(0, -height)
    # reposition the origin
    translate(width/2, height/2)
```

The preceding code structures an animated sketch by using `setup()` and `draw()` functions with two (pale blue) lines that intersect at the center of the display window. The y-axis is flipped, so y-coordinates decrease as you move downward; I'll elaborate on why I did that soon. The final `translate()` function shifts the coordinate system so that the origin (0, 0) sits in the center of the display window. This means that the x-coordinate for the left edge of the display window is −400, and the x-coordinate for the right edge is 400. The y-coordinate for the top edge is 300; for the bottom edge, it's −300 (Figure 9-10). The modified coordinate space, with its flipped y-axis, now behaves like a regular *Cartesian plane*, with four quadrants that allow you to plot any x-y coordinates ranging between (−400, −300) and (400, 300).

You've likely encountered this system in math classes before, which is why I've set up the coordinate space this way. You'll use it as a platform to experiment with elliptical and wave motion, but first, you may require a brief refresher on trigonometric functions.
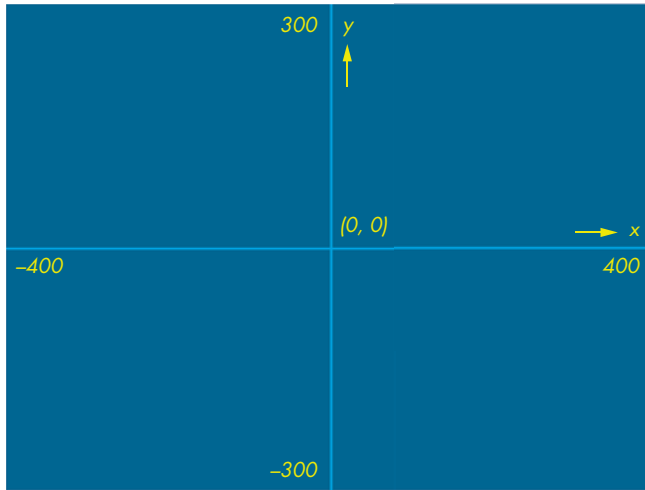
Figure 9-10: The Cartesian plane with four quadrants

An Introduction to Trigonometric Functions

Sine, *cosine*, and *tangent* are three common *trigonometric functions*. These are mathematical (as opposed to programming) functions, but you can use them in Python thanks to Processing's built-in trig functions. Sin, cos, and tan—as they are often abbreviated—are based on ratios obtained from a right triangle (Figure 9-11). A *right triangle* (or *right-angled triangle*) has one angle that measures exactly 90 degrees, usually denoted by a small square. The θ symbol, *theta*, is commonly used to represent an unknown angle.
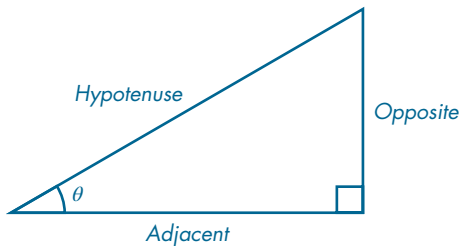
Figure 9-11: A right triangle

You can calculate the size of theta if you know the lengths of any two sides of this triangle. Depending on the lengths you have, you'll use either sin, cos, or tan for the calculation. *SOHCAHTOA*, pronounced phonetically as *so-ka-toe-uh*, is a handy mnemonic device to help you remember the following trigonometric ratios:

SOH $\sin(\theta)$ = opposite / hypotenuse

CAH $\cos(\theta)$ = adjacent / hypotenuse

TOA $\tan(\theta)$ = opposite / adjacent

As an example, if you know the length of the opposite and hypotenuse in Figure 9-11, you can find angle theta by using $\sin(\theta)$. If you know the length of the adjacent and hypotenuse, use $\cos(\theta)$. You can also rearrange these equations to find the length of an unknown side in cases when you know theta and one length. I'll return to this point shortly.

You'll apply sin and cos to a simple example to determine an x-y coordinate along the perimeter of a circle. To begin, draw a circle with its center positioned at (0, 0) with a radius of 200. Add a line starting at (0, 0) that's the same length as the circle radius and rotated 1 radian:

```
. . .
radius = 200
theta = 1

def draw():
    . . .
    circle(0, 0, radius*2)
    stroke('#FFFFFF')
    pushMatrix()
    rotate(theta)  # approximately 57.3 degrees
    line(0, 0, radius, 0)
    popMatrix()
```

The code renders the circle in a pale blue outline. A white line the length of the radius extends from the center of the circle to its perimeter; this forms an angle of 1 radian (equal to roughly 57.3 degrees), as labeled in Figure 9-12. Notice that the `rotate()` function applies counterclockwise to the line because the y-axis is inverted. The task is to work out the x-y coordinate for the point where the white line connects to the circle perimeter, labeled A. The other yellow markings reveal the right triangle upon which you'll base your calculations.
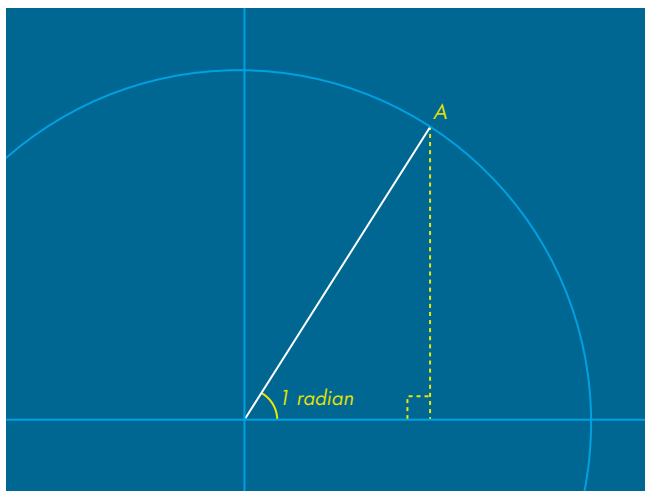
Figure 9-12: You'll find the x-y coordinate for the point labeled A.

Observe that the y-coordinate for point A is equal to the length (or height) of the opposite side. You know the angle (`theta` variable) and the length of the hypotenuse (`radius`), which you can use to calculate the length of the opposite. Recall that the *SOH* in *SOHCAHTOA* stands for *sin(θ) = opposite / hypotenuse.*

You have the values for θ and hypotenuse, so rearrange the equation to isolate opposite: *opposite = sin(θ) × hypotenuse.*

If you substitute the placeholders with the variable names in your program, this is `y = sin(theta) * radius`.

To calculate the x-coordinate for point A, you need to find the length (or width) of the adjacent side. Recall that the *CAH* in *SOHCAHTOA* stands for *cos(θ) = adjacent / hypotenuse,* which you can rearrange as `x = cos(theta) * radius`.

Add the following code to the end of your `draw()` function:

```
. . .
# white dot
noStroke()
fill('#FFFFFF')
x = cos(theta) * radius
y = sin(theta) * radius
circle(x, y, 15)
```

The `cos()` and `sin()` functions return floating-point values ranging from –1 to 1 for various values of `theta`. Processing's trig functions work with radians, so there's no need to convert the `theta` argument to degrees. In this case, theta is equal to 1 radian, and the `cos()` and `sin()` functions return values of 0.54 and 0.84, respectively (rounded to two decimal places). When you multiply 0.54 and 0.84 by the `radius` value of 200, you get an x-y coordinate of (108, 168). The `circle(x, y, 15)` function renders a white dot by using this x-y coordinate pair. Run the sketch to confirm the position of the white dot at point A, where the white line connects to the circle boundary.

You can adjust the theta value to move the white dot to different points along the perimeter of the pale blue circle. To position the dot at 90 degrees, directly above the origin, use `theta = HALF_PI`; for 180 degrees, use `theta = PI`; and so forth. A `theta` value of `TAU` brings you back around to the starting point, visually indistinguishable from a dot at `theta = 0`. If theta is greater than `TAU`, there's a wraparound effect. In other words, `cos(TAU+1)` is equivalent to `cos(1)`.

The next task is to get the dot moving. You don't need the white line anymore; remove it by deleting the lines starting from `pushMatrix()` up to and including `popMatrix()`.

Circular and Elliptical Motion

You'll begin by moving the dot along a circle perimeter (a circular motion), and you'll create a user-defined function for handling the necessary math. You'll then use this same function to create a spiral variant of the circular

motion. Once you have the circular and spiral motions working, you'll define a new function for elliptical motion. Figure 9-13 depicts examples of each motion.
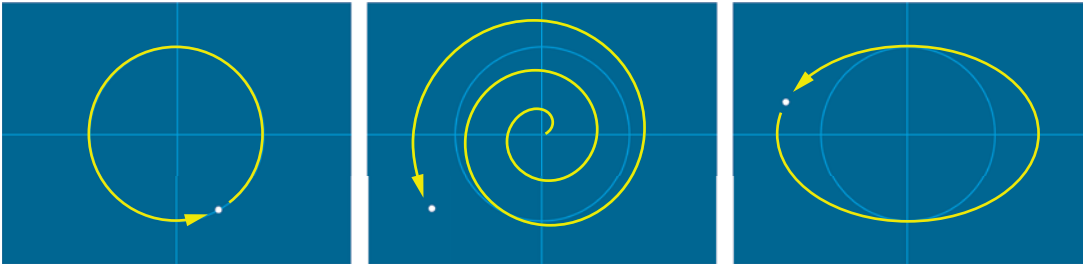
Figure 9-13: Circular (left), spiral (middle), and elliptical (right) motion

Circles

Recall that the size of angle theta, stored in a variable named `theta`, governs the position of the white dot. To make the dot move along the perimeter of the circle in a counterclockwise direction, add code to increment theta each time the draw function executes. Include a `period` variable to control the increment size:

```
. . .
period = 2.1

def draw():
    global theta
  ❶ theta += TAU / (frameRate * period)
    . . .
```

At the default `frameRate` of 60 fps, with a `period` of 2.1 seconds, the theta increment is equal to approximately 0.05 ❶. This means your angle extends 0.05 of a radian with each new frame. Run the sketch to test this out. The white dot should take about 2.1 seconds to complete a lap of the circle perimeter.

The larger the value you add to theta, the faster the dot will move. Subtracting from theta moves the dot in the opposite direction (clockwise).

Define a new function named `circlePoint()` for calculating points along the perimeter of a circle. In your `draw()` function, substitute the x and y lines with a `circlePoint()` function call:

```
def circlePoint(t, r):
    x = cos(t) * r
    y = sin(t) * r
    return [x, y]❶


. . .
def draw():
    . . .
```

```
#y = sin(theta) * radius
#x = cos(theta) * radius
x, y❷ = circlePoint(theta, radius)
circle(x, y, 15)
```

The `circlePoint()` definition includes two parameters: `t` for theta (the angle) and `r` for the radius. Because the function must calculate the x- and y-coordinates for some point along a circle perimeter, it needs to return two values. Use a list to return more than one value ❶; you could also use a dictionary (or a tuple).

When you call the function, Python can *unpack* the list values and assign them to multiple variables. To invoke this unpacking behavior, provide a corresponding variable for each list item, separating each variable with a comma. In this case, the function returns a list of two values, which are assigned to variables `x` and `y` ❷. Alternatively, you could assign the list to a single variable using something like `a = circlePoint(theta, radius)`, but then you'd have to refer to `x` and `y` by using `a[0]` and `a[1]`, respectively, which isn't as neat or descriptive.

Spirals

For an outward spiral motion (the center image in Figure 9-13), you can use a radius value that increases over time. Here's an example:

```
. . .
x, y = circlePoint(theta, frameCount)
circle(x, y, 15)
```

Recall that `frameCount` is a system variable containing the number of frames displayed since starting the sketch. The radius argument (the `frameCount`) begins at 0 and grows larger as the animation progresses, causing the dot to move outward in a spiral motion. The dot gains speed as it moves away from the center of the display window because each full rotation maintains the same period, regardless of the `circlePoint()` radius. In other words, the dot must cover a larger distance in the same time, so it moves faster.

Ellipses

For an elliptical motion, you need two radii: one for the horizontal axis and one for the vertical axis. These radii control the width and height of the ellipse shape that guides the white dot's trajectory (see the right image in Figure 9-13). Define a new `ellipsePoint()` function with parameters for an angle, horizontal radius, and vertical radius:

```
def ellipsePoint(t, hr, vr):
    x = cos(t) * hr
    y = sin(t) * vr
    return [x, y]
. . .
```

The function body is similar to that of the `circlePoint()` function. The difference is that you multiply the x and y values by the `hr` (horizontal-radius) and `vr` (vertical-radius) parameters, respectively.

The following `ellipsePoint()` function call makes the dot move in an elliptical motion:

```
. . .
x, y = ellipsePoint(theta, radius*1.5, radius)
circle(x, y, 15)
```

The `ellipsePoint()` function's second argument (horizontal radius) is larger than the third argument (vertical radius), so the resulting ellipse is wider than it is tall.

Sine Waves

A *sine wave* is a geometric waveform that repeats itself periodically, like a continuous chain of S-shaped curves connected end to end. This waveform features in many mathematical and physical applications. For example, you can use sine waves to model musical tones, radio waves, tides, and electrical currents.

The shape of a sine wave is formed using a `sin()` function. Figure 9-14 depicts a yellow sine wave.
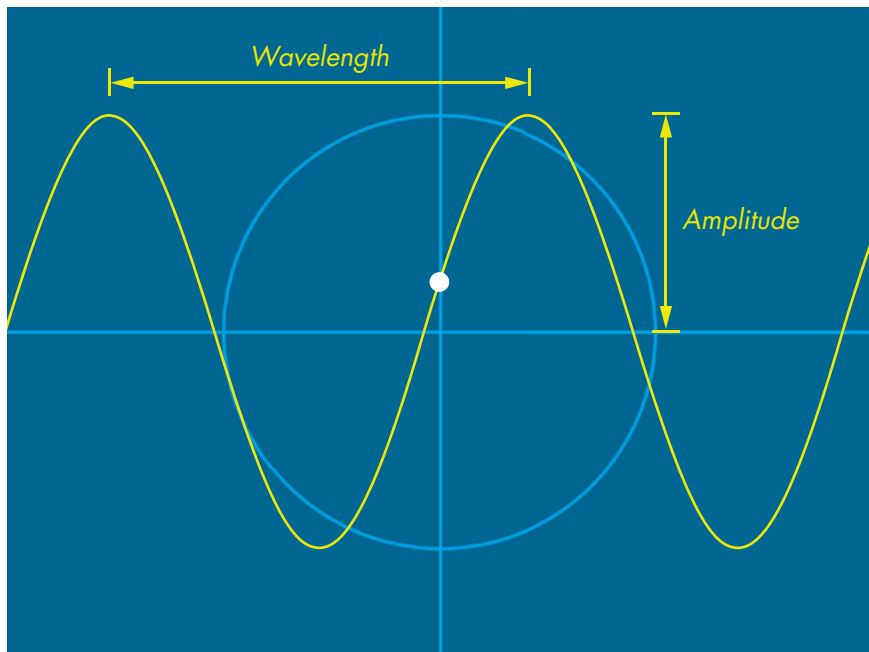
Figure 9-14: A sine wave

The *wavelength* is the length of one complete cycle, measured as the distance from crest to crest (or trough to trough). Wavelength is related to period, but period is a reference to time (taken to complete a cycle), and wavelength is a reference to distance.

The *amplitude* is the distance from the resting position (y = 0) to the crest. A wave with an amplitude of 0 would lie flat along the x-axis. You can determine that the yellow wave in Figure 9-14 has an amplitude of 200 by comparing it to the radius of the pale blue circle.

To simulate sine wave motion, add the following code to your *periodic_motion* sketch. This is the same as drawing a circle, but using a fixed x-coordinate:

```
. . .
def draw():
    . . .
    amplitude = radius
    y = sin(theta) * amplitude
    circle(0, y, 15)
```

The wave's amplitude is equal to the radius of the pale blue circle, although you can test any value you like. The y-coordinate for the white dot is calculated using sin(theta) multiplied by the amplitude; the x-coordinate is always 0. The result is a white dot that moves directly up and down from the origin.

Run the sketch and pay careful attention to how the dot is accelerating and decelerating, as if the wave shown in Figure 9-14 were passing through water with the dot floating on its surface. As the dot approaches a crest or trough, it begins to slow down, and then it accelerates after it makes a turn; it's moving fastest as it crosses the y-axis.

You can use this motion to draw a whole wave of moving dots or to simulate a weight hanging from a spring (Figure 9-15).
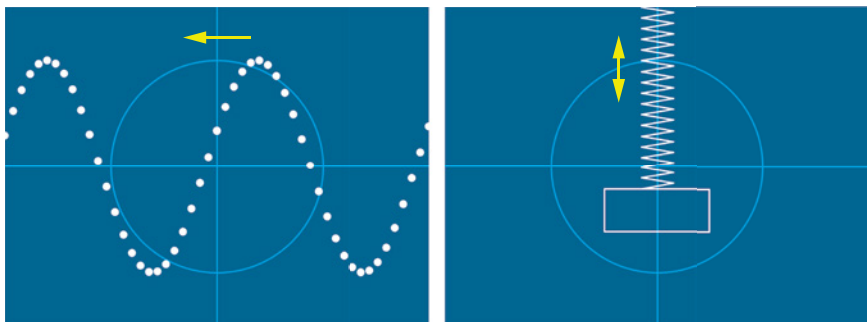
Figure 9-15: A wave of dots (left) and a weight hanging from a spring (right)

The code for each of these examples follows. You'll need to add it to the end of the draw() block of your *periodic_motion* sketch. You can add both code listings if you want to draw the spring and weight over the wave of dots, or instead replace one listing with the other.

Drawing a Sine Wave of Dots

Use a loop to draw a whole wave of dots. There are 51 dots in all, equally spread along the x-axis. Each dot has a different y-coordinate based on a `theta` value that's incrementally larger than the dot preceding it:

```
amplitude = radius

for i in range(51):
 ❶ f = 0.125 * 2
    t = theta + i * f
 ❷ x = -400 + i * 16
 ❸ y = sin(t) * amplitude
    circle(x, y, 15)
```

The loop draws 51 dots, beginning at an x-coordinate of –400, at x intervals of 16 pixels ❷. The y value for each dot is calculated using a `theta` value that's `0.125 * 2` of a radian (or 0.25) ❶ larger than the neighbor to its left. You can change this multiplier to `1` for a single wave that spans the width of the display window, leave it as `2` for two waves (as in Figure 9-15), make it `3` for three waves, and so forth. I've named the variable `f`, for *frequency*, which refers to the number of times an event repeats itself in a fixed time period.

Wavelength is inversely proportional to frequency, so as you increase the frequency, you decrease the wavelength (and the waves begin to look spikier). The wave motion travels from right to left, but the horizontal positions of the dots don't change.

Simulating a Weight Hanging from a Spring

Use a loop to draw the spring, which is a shape composed of vertices. The weight dangling on the end of the spring is a rectangle. Adjust the fill and stroke to draw outlines instead of filled shapes:

```
amplitude = radius
y = sin(theta) * amplitude
noFill()
stroke('#FFFFFF')
strokeJoin(ROUND)
bends = 35

beginShape()
for i in range(bends):
    vx = 30 + 60 * (i % 2 - 1)
    vy = 300 - (300 - y) / (bends - 1) * i
    vertex(vx, vy)
endShape()

rect(-100, y-80, 200, 80)
```

The tight corners of the spring's bends will produce sharp joints, which result in elongated "elbows." Processing clips these when they get too long and sharp, but jumping between *mitered* (sharp) and *beveled* (clipped) joints

makes the animation look bad. To prevent this, I've set the `strokeJoin` to `ROUND`. A loop is nested within the `beginShape()` and `endShape()` functions for plotting the zigzagging spring vertices.

Ordinarily, some energy is dissipated or lost in such a system, and the amplitude should decay over time. You could simulate this by reducing the (global) `radius` value every frame until it reaches 0, when the weight will come to a rest.

Now that you've learned how to return values from functions and incorporate trigonometry for elliptical and wave animation, let's look at a special curve created by combining waves.

Lissajous Curves

In this section, you'll create a function for drawing Lissajous curves controlled by arguments. A *Lissajous curve*—named after French physicist Jules Antoine Lissajous—is formed by combining x- and y-coordinates from two waves.

You can create these curves mechanically by setting up a Y-shaped pendulum with a sand-filled cup hanging at the end of it. As the cup swings about, sand drains through a hole at the bottom, drawing a curve. Figure 9-16 shows an example of this device (left) and an image of a curve drawn with sand (right). The point labeled *r* indicates where the pendulum merges into a single string. The ratio of the upper to lower section of the pendulum, and the angle and power of your initial swing, determine the shape of the resulting curve.
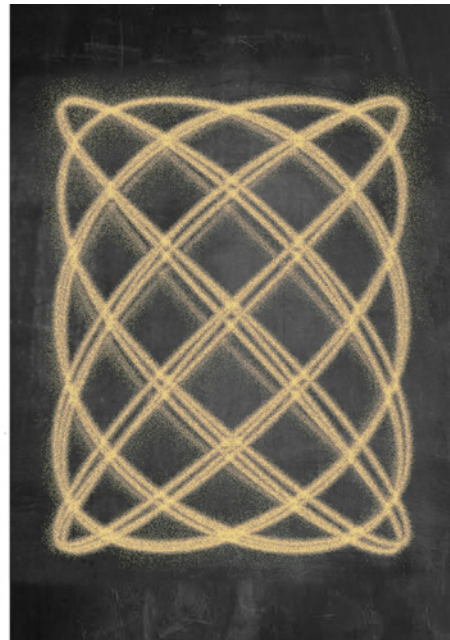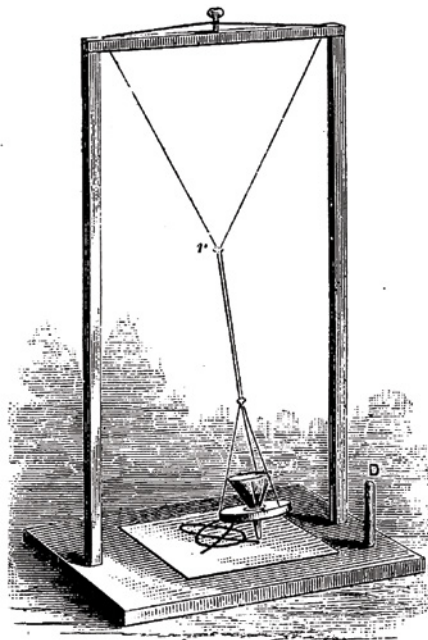
Figure 9-16: Blackburn's Y-shaped pendulum, from Sound *by John Tyndall, 1879 (left), and a Lissajous curve drawn with sand (right)*

To begin, suppose that you have two circles of different sizes (Figure 9-17). Circle A has a radius labeled A that is 200 units, and Circle B has a radius labeled B that is 100 units.
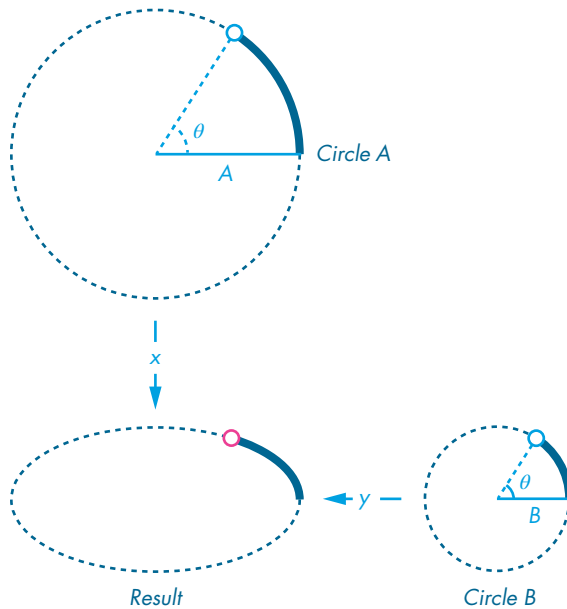
Figure 9-17: Combining x and y values from different circles to form an ellipse

The Result ellipse (lower left) is formed by using x-coordinates from Circle A and y-coordinates from Circle B. The ellipse turns out as wide as Circle A and as tall as Circle B. The math for this is relatively simple and uses what you already know about drawing ellipses with trigonometric functions.

To find the x-y coordinate for any point along the perimeter of the Result ellipse, you use the following:

$x = cos(\theta) \times A$

$y = sin(\theta) \times B$

Create a new sketch, save it as *lissajous_curves*, and add the following code to recreate the ellipse from Figure 9-17:

```
def lissajousPoint(t, A, B):
    x = cos(t) * A
    y = sin(t) * B
    return [x, y]

def setup():
    size(800, 600)
    frameRate(30)
    background('#004477')
    fill('#FFFFFF')
```

```
    noStroke()

theta = 0
period = 10

def draw():
    global theta
    theta += TAU / (frameRate * period)
    # flip the y-axis and reposition the origin
    scale(1, -1)
    translate(width/2, height/2-height)

    x, y = lissajousPoint(theta, 200, 100)
    circle(x, y, 15)
```

The drawing space is set up like your preceding sketch. You have an inverted y-axis, and the origin is shifted to the center of the display window. The theta value increments by approximately 0.01 each frame, which serves as the first argument in the lissajousPoint() function call. Right now, this function performs exactly the same operation as the ellipsePoint() function in your *period_motion* sketch—the only difference is the naming of the function and its variables.

Notice that there's no background() call within the draw() section of the code, so Processing won't clear each frame. Because of this, the moving white dot forms a continuous line. Run the sketch; it should draw a complete ellipse in a counterclockwise motion (Figure 9-18).

Figure 9-18: Drawing an ellipse by using the ellipsePoint() function

When theta reaches τ radians (~6.28), the oval is complete, and Processing continues to draw over the existing line. Even though the animation might appear complete, the dot is still moving along the perimeter.

The next step is to modify the lissajousPoint() function so that it can draw Lissajous curves (as opposed to ellipses). But first, consider what's happening here in terms of waves. Study Figure 9-19, which represents each circle as a wave, and take note of how the dots on each wave control the position of the dot along the ellipse's perimeter.
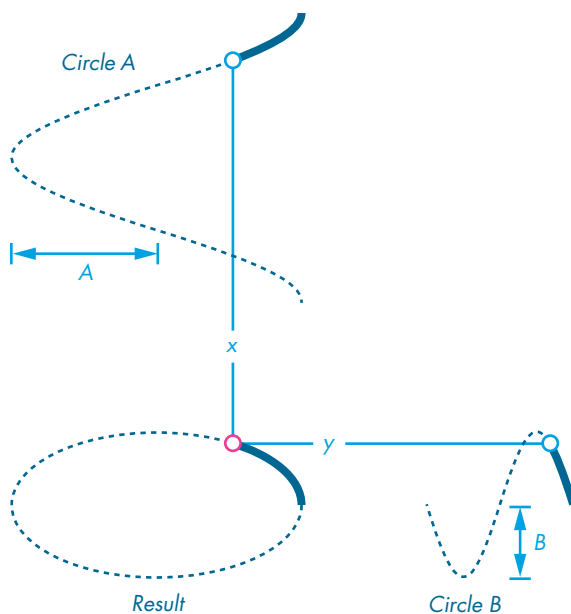
Figure 9-19: Circle A and Circle B represented in wave form

Figure 9-19 presents the x-coordinates of Circle A as a cosine wave that oscillates between –1 and 1, which is scaled by the circle radius (the wave amplitude) of A. Similarly, the y-coordinates of Circle B are presented as a sine wave with an amplitude of B.

In Figure 9-20, you can see how dots move along the waves to form the ellipse shape.
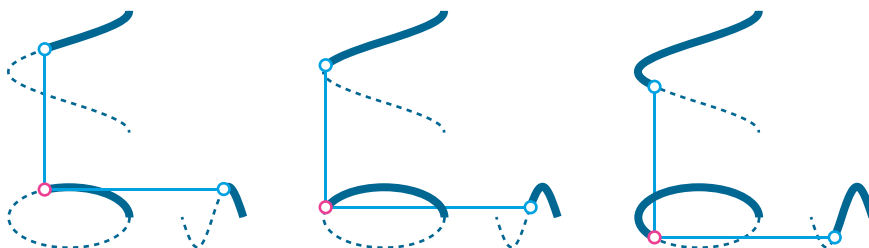
Figure 9-20: Theta = 2 (left), theta = 3 (middle), theta = 4 (right)

Currently, the frequencies of both waves match. In other words, it takes the same amount of time for each wave to complete a single cycle. The result is an ellipse.

Lissajous curves occur when the wave frequencies differ. In Figure 9-21, the frequency of the Circle B wave is twice that of the Circle A wave. The dot following the Circle B wave must complete two cycles in the same amount of time that the Circle A dot will complete one. The *a* and *b* values (lowercase) represent a frequency of 1 and 2, respectively.
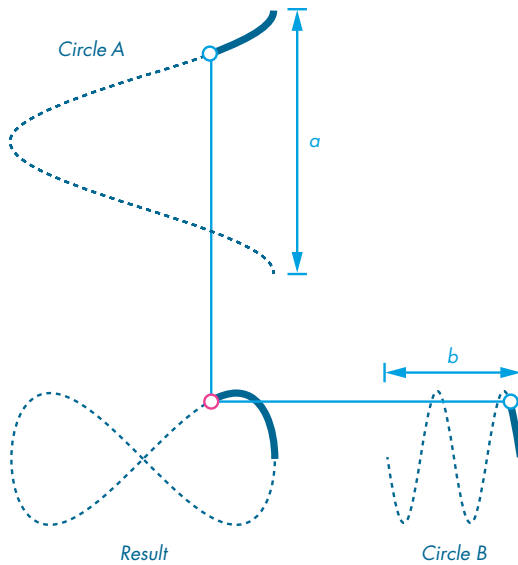
Figure 9-21: The Circle B wave has a frequency
twice that of Circle A.

Frequencies *a* and *b* could be 3 and 6, 40 and 80, or 620 and 1,240. Any pair of numbers with a ratio of 1:2 will produce a ∞ shape. This will be important when you return to writing the code. You can think of this in another way as well: in Figure 9-17, the Circle B dot must always complete two journeys around the perimeter in the same amount of time that the Circle A dot completes one.

Figure 9-22 shows how the dots move along the waves to form the Lissajous curve.

Adapt your `lissajousPoint()` definition, adding a parameter for frequency *a* and frequency *b*. Use these two parameters as multipliers for theta (t) in your x and y lines, respectively:

```
def lissajousPoint(t, A, B, a, b):
    x = cos(t * a) * A
    y = sin(t * b) * B
    return [x, y]
    . . .
```
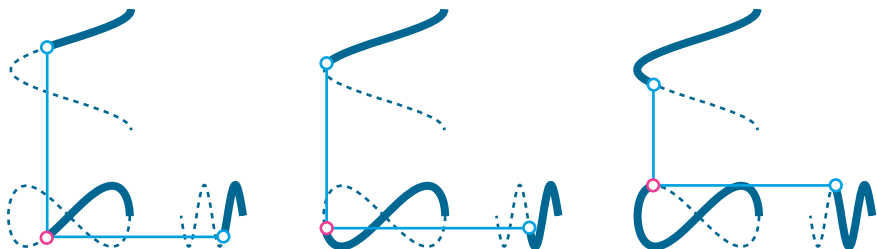

Figure 9-22: From left to right: theta = 2; theta = 3; theta = 4

Now, add arguments for parameters `a` and `b` to your function call:

```
x, y = lissajousPoint(theta, 200, 100, 1, 2)
```

Run the sketch and watch Processing draw a Lissajous curve (Figure 9-23).
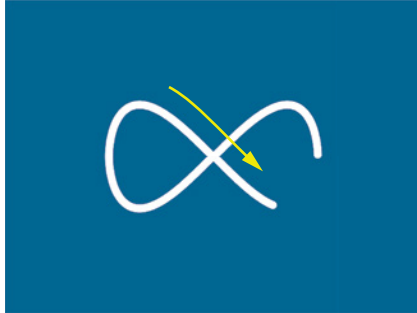
Figure 9-23: Drawing a Lissajous curve by using the `lissajousPoint()` function

The `a` and `b` arguments determine the number of horizontal and vertical "lobes" in the Lissajous curve. Recall that it's the ratio that matters, so `1, 2` will produce the same curve as `5, 10`. However, the latter pair will complete drawing the curve in less time, and even larger numbers will create discernible spacing between the dots (that would otherwise form a solid line). Figure 9-24 shows the results of a few `a`, `b` arguments. Try experimenting with other numbers.
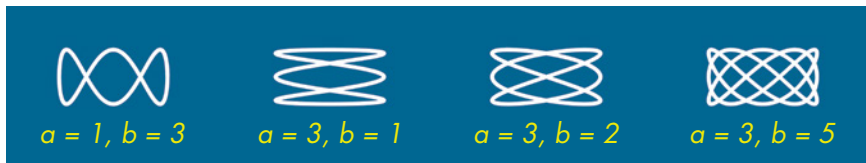
Figure 9-24: Drawing Lissajous curves using different a, b arguments

You can create intriguing visual patterns by moving shapes, points, and lines around with trigonometric functions. Simply experimenting, with no predefined idea of what you want to accomplish, can lead to impressive visual results. Think of this approach to coding like a musical jam session, where instrumentalists improvise until they stumble upon something that sounds good.

The next task uses Lissajous curves and a `line()` function for animated patterns, which should provide some interesting ideas for you to riff off.

Creating Screensaver-Like Patterns with Lissajous Curves

In Chapter 6, you programmed a simple DVD screensaver; now let's create a more elaborate one using Lissajous curves. The original purpose

of a screensaver was to "save" your screen. Older cathode-ray tube (CRT) monitors were susceptible to *burn-in*: if you displayed the same graphic in the same position for too long, it would leave a permanent "ghost" image. Modern displays aren't susceptible to burn-in, but many people still use screensavers because they look cool.

You'll use your `lissajousPoint()` function to create a pattern inspired by popular screensaver designs. Figure 9-25 shows the final result with lines and colors morphing smoothly as the pattern twists about the screen.
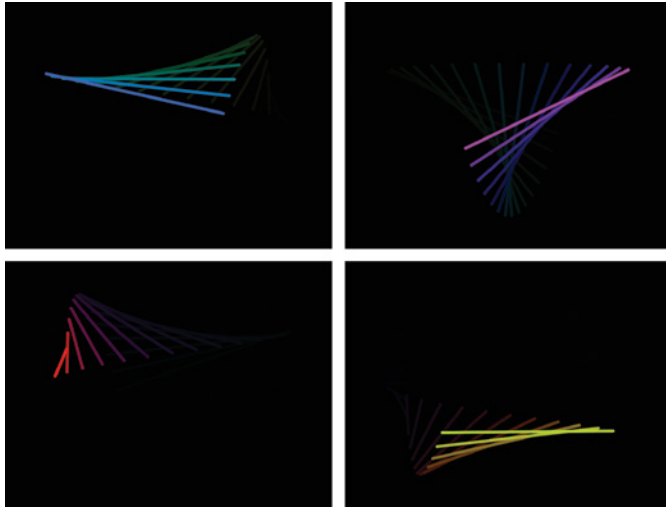
Figure 9-25: An animated pattern based on Lissajous curves

This movement relies on two Lissajous curves, using a `line()` function to draw a straight line between the leading tip of each curve. Figure 9-26 illustrates how this works.
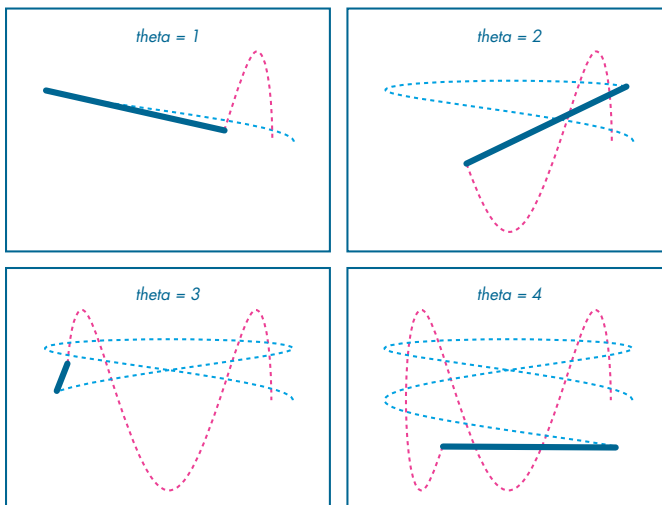
Figure 9-26: Drawing a straight line between two Lissajous curves

Of course, you don't see the curves, just the straight line, but it's two `lissajousPoint()` calls that are calculating the x-y coordinate for your `line()` function. When theta reaches τ radians, the Lissajous curves are complete and the motion repeats itself.

Add the following code to the end of the `draw()` function in your *lissajous_curves* sketch:

```
    . . .
❶ for i in range(10):
        # curves
        t = theta + i / 15.0
        x1, y1 = lissajousPoint(t, 300, 150, 3, 1)
        x2, y2 = lissajousPoint(t, 250, 220, 1, 3)
        # background color
    ❷ fill(0x55000000)
        noStroke()
        rect(-width/2, -height/2, width, height)
        # line
        colorMode(HSB, 360, 100, 100)
    ❸ h = (frameCount + i * 15) % 360
        strokeWeight(7)
        stroke(h, 100, 100)
        line(x1, y1, x2, y2)
```

The loop will draw 10 lines ❶ in all—one solid line leading a trail of nine lines that gradually fade behind it. You use two `lissajousPoint()` functions, one for each curve (that together define the x-y coordinate for each end of the line). With each iteration, Processing draws a semiopaque black square that spans the entire display window, dimming the lines of previous iterations.

To define a semiopaque color, you use Processing's `0x` notation ❷. The hexadecimal value is expressed with a leading `0x`, without quotes, using eight hexadecimal digits. The first two digits define the *alpha* (transparency) component; for example, `11` is highly transparent, and `EE` highly opaque. This example uses `55`, somewhere in between, but nearer the transparent side. The remaining six characters are your standard RGB hexadecimal mixture, in this case black (`000000`). For the stroke color, set the `colorMode()` to `HSB` (see "Color Modes" on page 14). For the first 360 frames, you can use `frameCount` to shift the hue value a single degree per frame. However, `frameCount` will soon exceed 360, so you use a modulo operation to "wrap around" back to 0 ❸.

Run the sketch to observe the output.

NOTE *Drawing so many semiopaque black rectangles over the display window each frame is a demanding operation for Processing to perform. If your computer is struggling, try setting a lower frame rate, or reducing the `for` loop iterations from 10 to a more manageable value.*

Try different `lissajousPoint()` arguments, or add new curves and lines; maybe even try to connect three lines between three curves for morphing triangles. Keep experimenting to see what you come up with.

Summary

In this chapter, you've learned how to define your own functions, which reduce repetition and help you structure more modular programs. Remember that well-named functions will make your code easier to read and understand, for yourself and anybody else dealing with it.

You can add parameters to any function to make it more versatile, and the function call will include different arguments that correspond to those parameters to control how it works. You can call a function by using positional and/or keyword arguments. For optional arguments, you can define parameters that include default values for Python to fall back on.

You can also define functions that return values, which means you can use a function to process data and hand back a result to the function caller. If a function returns a value, you can assign it to a variable. Additionally, you can wrap a function around an argument to process and return a value for another function.

This chapter also introduced trigonometry concepts and how to use them to simulate periodic motion. You learned about built-in Processing trig functions, like `sin()` and `cos()`, which you used to draw circles, spirals, ellipses, sine waves, and Lissajous curves. Experiment with trigonometry to generate compelling patterns and movements like those you see in some screensavers.

In the next chapter, you'll write *classes*, which you will use to create *objects*. These techniques enable you to structure your code more efficiently, especially for larger, more complex programs, by modeling your programs around real-world objects. You'll also learn about *vectors* for programming motion.

10

OBJECT–ORIENTED PROGRAMMING AND PVECTOR

Object-oriented programming (OOP) deals with data structures known as *objects.* You create new objects from a class, and you can think of a class as an object template, composed of a collection of related functions and variables. You define a class for each category of objects you want to work with, and each new object will automatically adopt the features you define in its class. OOP combines everything you've learned so far, including variables, conditional statements, lists, dictionaries, and functions. OOP adds a remarkably effective way to organize your programs by modeling real-world objects.

You can use classes to model tangible objects, like buildings, people, cats, and cars. Or, you can use them to model more abstract things, like bank accounts, personalities, and physical forces. Although a class will define the general features of a category of objects, you can assign unique attributes

to differentiate each object you create. In this chapter, you'll apply OOP techniques to program an amoeba simulation. You'll learn how to define an amoeba class, and how to "spawn" varied amoeba from it.

You'll program amoeba movement by simulating physical forces. For this, you'll use a built-in Processing class named `PVector`. The `PVector` class is an implementation of *Euclidean vectors* that includes a suite of methods for performing mathematical operations, which you'll use to calculate the position and movement of each amoeba.

To better manage your code, you'll learn how to split your program into multiple files. You can then switch between the files that make up your sketch by using tabs in the Processing editor.

Working with Classes

A *class* is like a blueprint for an object. As an example, consider a `Car` class that might specify, by default, that all cars have four wheels, a windshield, and so on. Certain features, like the paint color, can vary among individual cars, so when you create a new car object by using the `Car` class, you get to select a color. Such features are called *attributes*. In Python, attributes are variables that belong to a class. You can decide which attributes have predefined values (the four wheels and windshield) and which are assigned when you create a new car (the paint color).

In this way, you can create multiple cars, each a different color, using a single class. Figure 10-1 illustrates this concept. The `Car` class includes attributes to describe the paint color, engine type, and model of each car.
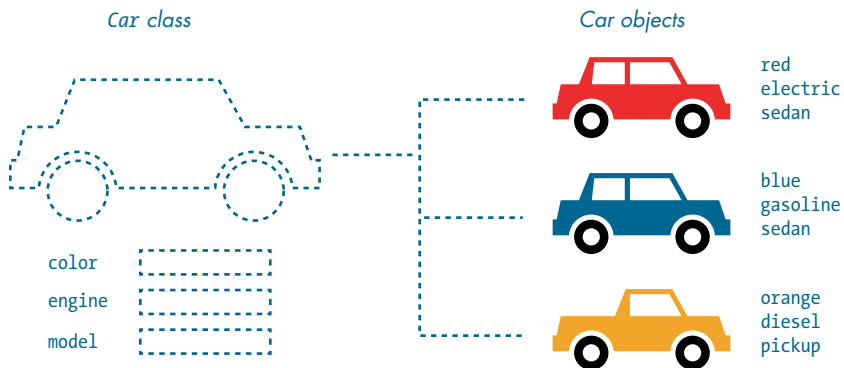
Figure 10-1: The `Car` class serves as a blueprint for car objects.

Drivers control a vehicle by steering, accelerating, and braking. So in addition to attributes, your `Car` class can include definitions for performing those actions, referred to as *methods*. In Python, methods are functions that belong to a class that define the operations or activities it can perform.

Now, let's define an `Amoeba` class that includes a set of attributes and methods for controlling the appearance and behavior of amoeba objects. You'll use that class to create many amoebas. Figure 10-2 depicts the final result of the amoeba simulation that you're working toward.
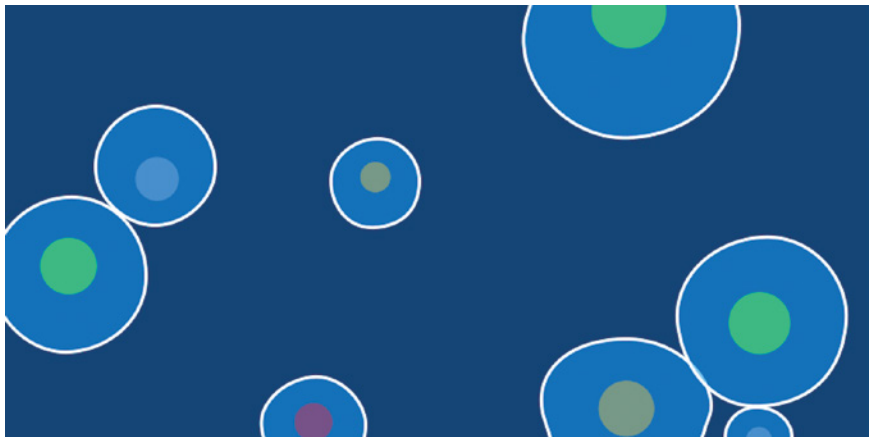
Figure 10-2: A screenshot of the complete amoeba simulation

The amoebas will wobble and distort as they move about the display window. This is not a scientifically correct representation of amoebas, but it should look pretty cool. As an extra challenge, you'll add collision-detection code to prevent them from passing over or through one another. You'll begin with a basic `Amoeba` class definition, and then add attributes and methods as you progress through the task.

Defining a New Class

In Python, you define a class by using the `class` keyword. You may name a class whatever you like, but as with variable and function names, you're limited to alphanumeric and underscore characters. Because you cannot use space characters, the recommended naming convention for classes is *UpperCamelCase*, in which the first letter of each word begins with a capital letter, starting with the first word.

To begin, your `Amoeba` class won't do much else than print a line to the console. Start a new sketch and save it as *microscopic*. Define a new `Amoeba` class:

```
class Amoeba(object):

    def __init__(self):
        print('amoeba initialized')
```

The `class` keyword defines a new class. Here the class name is `Amoeba`, and it's followed by `object` in parentheses, and a colon.

If you run the sketch, nothing interesting should happen, and the console will be empty.

NOTE   *Python 2 has "old-style" and "new-style" classes. You'll want to use the new style, which is why I include `object` in parentheses. This isn't required in Python 3, because its classes are always new style. That said, it won't make a difference if you happen to include the `object` part in your Python 3 programs.*

Functions that you define within the body of a class are referred to as *methods*. The `Amoeba` class includes a definition for a special method named `__init__` (with two underscores at either end). This method is one of a selection of *magic methods* that start and end with two underscores that you won't invoke directly. I'll get into more detail about the `__init__()` method (and the `self` parameter) soon. For now, all you need to know is that Python runs the `__init__()` method automatically for each new amoeba you create. You use this method to set up your attributes and execute code at the time of object creation.

Creating an Instance from a Class

To *instantiate* an amoeba, you call the `Amoeba` class by name and assign it to a variable—as you would a function that returns a value. *Instantiate* is a fancy way of saying *create a new instance*, and an *instance* is synonymous with *object*.

NOTE   *You'll often hear the terms* object *and* instance *used interchangeably. Correctly speaking, you create amoeba* objects *from the `Amoeba` class. A given amoeba is an* instance *of the `Amoeba` class.* Instance *emphasizes the distinct identity of a particular amoeba.*

Add a line to create a new instance from your `Amoeba` class and assign it to a variable named `a1`:

```
class Amoeba(object):

    def __init__(self):
        print('amoeba initialized')

a1 = Amoeba()
```

When you run the sketch, Python creates a new `Amoeba()` instance. This will automatically invoke the `__init__()` method. You can use the `__init__()`

method to define attributes and assign values to them, which you'll do shortly. This method can also include other instructions to initialize the amoeba, as in this case, a print() function. When you run the sketch, the console should display a single amoeba initialized message.

Adding Attributes to a Class

You can think of attributes as variables that belong to an object. And just like a variable, an attribute can contain any data you like, including numbers, strings, lists, dictionaries, and even other objects. For example, a Car class might have a string attribute for the model name and an integer attribute for top speed.

In your Amoeba class, you'll add three attributes to hold numbers for an x-coordinate, y-coordinate, and diameter; you'll assign values to those attributes when you instantiate the new amoeba. The syntax resembles that used to pass arguments to a function: the parentheses of the __init__() method contain your list of corresponding parameters.

Make the following changes to your code to accommodate an x, y, and diameter value for each new amoeba:

```
class Amoeba(object):

    def __init__(self, x, y, diameter):
        print('amoeba initialized')

a1 = Amoeba(400, 200, 100)
```

The __init__() method already includes a parameter, self; this is required, and it's always the first parameter. The self parameter provides access to instance-specific values, like an x value of 400 for amoeba a1 (but more on how that works shortly). The x, y, and diameter are added as the second, third, and fourth parameters. I've added corresponding arguments to the a1 line. Notice, however, that I provide only three arguments and nothing for the self parameter. Figure 10-3 depicts how these positional arguments match up, starting from the second parameter in the __init__() method.

Figure 10-3: Don't provide an argument for the self parameter.

You can also use keyword arguments (and specify default values for parameters), but I'll stick to positional arguments throughout this task.

NOTE *If you pass the wrong number of arguments to __init__() or any other class method, Python will display an error message. But this error message can confuse many beginners. As an example, you can try creating a new Amoeba class with four arguments by using Amoeba(400, 200, 100, 777). Run the sketch, and the Python error message*

*will report that the __init__() method takes exactly four arguments, claiming that
you've given five. This is because the self parameter makes it four arguments, but
Python passes that value implicitly, leaving just three arguments for you to provide.
Keep this in mind when you're debugging OOP code.*

When you pass values to your __init__() method, it won't automatically
store them for you. For this, you need attributes, which are like variables for
objects. Assign the x, y, and diameter parameters to new attributes. Each attri-
bute begins with a prefix of self, followed by a dot, then the attribute name:

```
class Amoeba(object):

    def __init__(self, x, y, diameter):
        print('amoeba initialized')
        self.x = x
        self.y = y
        self.d = diameter

a1 = Amoeba(400, 200, 100)
```

Notice that you assign diameter to self.d. Your attribute names need not
match your parameter names.

At this point, I can explain more about the self parameter. I've men-
tioned that self is an instance-specific reference. In other words, the self.d
value of 100 belongs to amoeba a1. Each amoeba instance will possess its own
set of self.x, self.y, and self.d values. For example, I might add another
amoeba, a3, with different values:

```
a3 = Amoeba(600, 250, 200)
```

This will come in handy later when you add multiple amoebas to the
simulation. Figure 10-4 provides a conceptual diagram of your Amoeba class
and three possible instances.

Next, you'll learn how to access the x, y, and d values for amoeba a1 via
the a1 instance. You'll use those values to draw the amoeba in the display win-
dow, resembling the one depicted in the upper right corner of Figure 10-4.
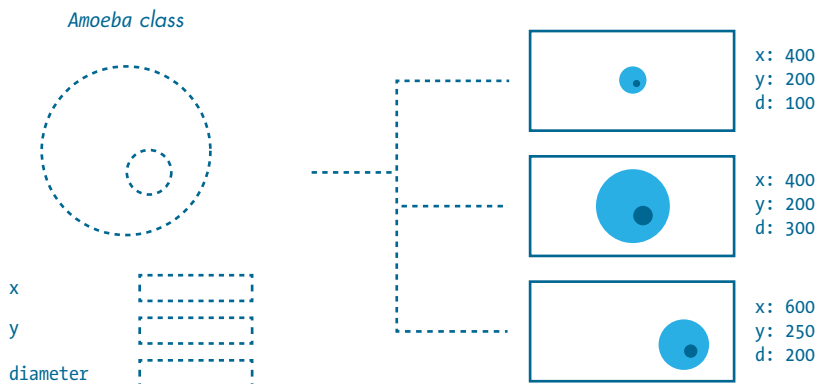
Figure 10-4: Your Amoeba class and three instances

Accessing Attributes

To access attributes, you use *dot notation*. For the `a1` instance, you can access the x, y, and d attributes as `a1.x`, `a1.y`, and `a1.d`, respectively. This is the instance name (`a1`) followed by a dot, followed by the name of the attribute you want to access.

To get started, add this code to the end of your sketch, which draws a circle to represent amoeba `a1`:

```
. . .
def setup():
    size(800, 400)
    frameRate(120)

def draw():
    background('#004477')
    # cell membrane
    fill(0x880099FF)
    stroke('#FFFFFF')
    strokeWeight(3)
    circle(a1.x, a1.y, a1.d)
```

The display window is now 800 pixels wide by 400 pixels high. The high frame rate of 120 will help smooth the wobble animation you'll add to your amoeba later. A *cell membrane* separates an amoeba's interior from its outside environment, and here, I've given this a white stroke. The fill is a semi-opaque pale blue. For the x-coordinate (first argument) in the `circle()` function, Python checks the `a1` instance for the attribute `self.x`—in this case, it's equal to 400; the y-coordinate argument is equal to 200, and the diameter argument is equal to 100. The result (Figure 10-5) is a circle with a diameter of 100 pixels positioned in the center of the display window.
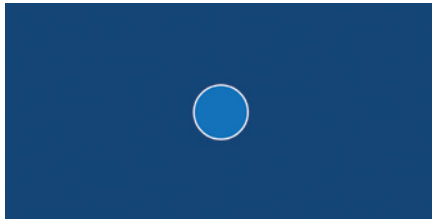
Figure 10-5: A circle (rudimentary amoeba) with a diameter of 100 pixels

So far, you've learned how to add arguments to your Amoeba class, which you assign to attributes when you instantiate an amoeba. In addition to those, your class can include attributes with predefined values.

Adding an Attribute with a Default Value

Think back to the car analogy. Every car rolls off the production line with an empty gas tank. The manufacturer may fill it before it's sold, but the tank always starts empty. For this, you decide to add an attribute to the Car

class—let's call it self.fuel. It has a predefined value of 0 for each new car object, but it'll fluctuate over the lifetime of the vehicle. It's redundant to specify by way of an argument that this should start at 0; instead, the Car class should automatically initialize the fuel attribute for you, setting it to 0 by default.

Let's return to the amoeba task. Every amoeba will include a nucleus with a predefined fill of red. To program this, assign a hexadecimal value (#FF0000) to an attribute named nucleus within the body of your __init__() method. There's no need to add another parameter to your __init__() definition, because you don't require the additional argument to specify the red fill:

```
    . . .
    self.x = x
    self.y = y
    self.d = diameter
    self.nucleus = '#FF0000'
. . .
```

Now, every amoeba you create has a nucleus attribute assigned a value of #FF0000.

Insert three new lines in your draw() function to render the nucleus beneath the cell membrane:

```
. . .
def draw():
    background('#004477')
    # nucleus
    fill(a1.nucleus)
    noStroke()
    circle(a1.x, a1.y, a1.d/2.5)
    # cell membrane
    . . .
```

The new lines set the fill and stroke, and then draw the nucleus by using a circle() function with a diameter that's 2.5 times smaller (a1.d/2.5) than that of the cell membrane, placing it in the center of the amoeba. Run the sketch to confirm that you see a mauve nucleus; it is technically red, but you see it through the pale blue, semi-opaque membrane.

You don't set the nucleus fill when you instantiate the amoeba, but that doesn't mean you're stuck with a red nucleus. You can modify the attribute values after you've created an amoeba.

Modifying an Attribute Value

Many attributes hold values that change as your program runs. To return to the car analogy, consider the fuel attribute mentioned previously with a value that's continually shifting as the gas tank fluctuates between full and empty. You can modify the value of any attribute directly via the instance by using the same dot syntax for accessing values.

Insert a line to change the nucleus fill for amoeba instance `a1`:

```
. . .
# nucleus
a1.nucleus = '#00FF00'
fill(a1.nucleus)
. . .
```

This sets the `nucleus` attribute to green, overwriting the default value of red. Run the sketch to confirm that you see a green nucleus showing through the semi-opaque membrane.

You can also modify an attribute by using a method, which I cover in "Adding Methods to a Class" on page 216.

Using a Dictionary for an Attribute

Recall that attributes can contain anything you like—numbers, strings, lists, dictionaries, objects, and so on. You'll use a dictionary attribute that holds a mix of string (hexadecimal) and floating-point values to group the nucleus properties.

Change your `nucleus` attribute to a dictionary that holds key-value pairs for a nucleus fill, x-coordinate, y-coordinate, and diameter. To vary the appearance of each amoeba, randomize those values:

```
class Amoeba(object):

    def __init__(self, x, y, diameter):
        print('amoeba initialized')
        self.x = x
        self.y = y
        self.d = diameter
        self.nucleus = {
          'fill': ['#FF0000', '#FF9900', '#FFFF00',
                   '#00FF00', '#0099FF'][int(random(5))],
          'x': self.d * random(-0.15, 0.15),
          'y': self.d * random(-0.15, 0.15),
          'd': self.d / random(2.5, 4)
        }
. . .
```

The `fill` key is paired with a hexadecimal value arbitrarily selected from a list of five colors. The nucleus color of each new amoeba is now chosen at random (although you may explicitly overwrite it afterward). The `x` and `y` keys are assigned randomized values proportional to the diameter of the cell membrane; you'll use those to position the nucleus somewhere within the boundary of the cell membrane, but not necessarily in the center. The diameter of the nucleus (`d`) is also proportional to the cell membrane and randomly varies for each instance.

Update your `draw()` code to work with these changes:

```
. . .
def draw():
```

```
background('#004477')
# nucleus
fill(a1.nucleus['fill'])
noStroke()
circle(
  a1.x + a1.nucleus['x'],
  a1.y + a1.nucleus['y'],
  a1.nucleus['d']
)
# cell membrane
. . .
```

The fill() and circle() arguments reference the relevant dictionary keys to style and position the nucleus.

Each time you run the sketch, Processing will generate a unique amoeba. Figure 10-6 depicts four results from four runs. Of course, it's possible (but unlikely) that Processing will produce the same or a similar selection of randomized values, and consecutive results might appear identical.
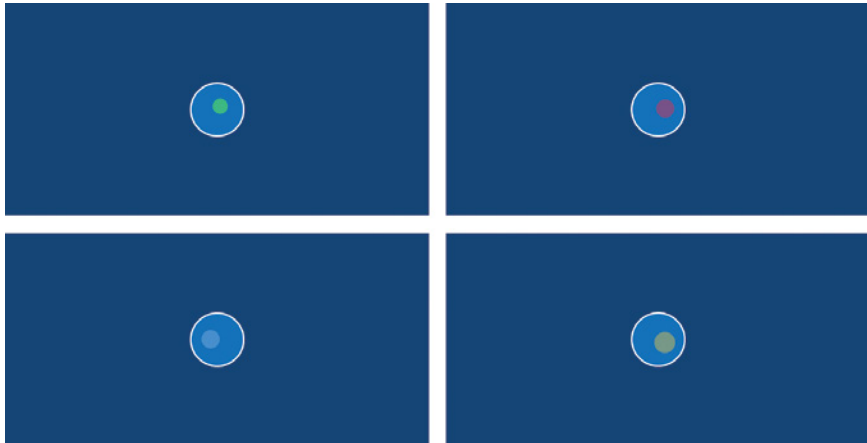
Figure 10-6: Each amoeba is generated using randomized nucleus values.

Now that you've set up the attributes to control the visual appearance of your amoeba, the next step is to add methods to animate it.

Adding Methods to a Class

Functions that you define within the body of a class are referred to as *methods*. To return to the car analogy, drivers can control a vehicle by using methods, such as steering, accelerating, and braking. You could also include a method for refueling. Methods typically perform operations by using an object's attributes. For example, an accelerate() and refuel() method will subtract from and add to a fuel attribute.

Another analogy for describing object-oriented programming uses parts of speech. It goes like this: if objects (cars) are nouns, and attributes (like paint color) are adjectives, then methods (steer, accelerate, brake, and refuel) are verbs.

You can name methods whatever you like, as long as you apply the same naming rules and conventions for functions. In other words, use only alphanumeric and underscore characters, camelCase or underscores instead of spaces, and so forth.

You'll create a new method to draw your amoeba for each frame. Currently, several lines in the draw() section of your code handle this operation. Move the nucleus and cell membrane code from the draw() function into the body of a new display() method, ensuring that your indentation is correct. Replace every a1 prefix with self in the display() method:

```
class Amoeba(object):
    . . .
    def display(self❶):
        # nucleus
        fill(self.nucleus['fill'])
        noStroke()
        circle(
          self.x + self.nucleus['x'],
          self.y + self.nucleus['y'],
          self.nucleus['d']
        )
        # cell membrane
        fill(0x880099FF)
        stroke('#FFFFFF')
        strokeWeight(3)
        circle(self.x, self.y, self.d)
. . .
def draw():
    background('#004477')
```

The self parameter in the definition ❶ provides the body of your display() method with access to your attributes, such as self.nucleus and self.x. The display() method accepts zero arguments, so the definition includes no further parameters.

Calling a Method

Once you've defined a method, you can use the same dot notation as for attributes to call the method and execute the code in that method's body—that is, the instance name followed by the method, separated by a dot. Of course, methods, like functions, include parentheses, and sometimes arguments too.

Add an a1.display() call to your draw() function to render amoeba a1:

```
. . .
def draw():
    background('#004477')
    a1.display()
```

You have no parameters (other than `self`) in your `display()` definition, so the method call takes no arguments. Run the sketch to confirm that it produces the same result as before (Figure 10-6).

Just like well-named functions, well-named methods make your code easier to read and understand, for yourself and anybody else dealing with it.

To get your amoeba wobbling, you'll define a new method that you call from within the `Amoeba` class. Additionally, this method will accept a few arguments.

Creating a Wobbly Amoeba

Amoebas distort and ripple, like balloons full of water. To replicate this not-quite-circular shape, you'll replace the cell membrane's `circle()` function with a shape formed using `bezierVertex()` functions. This is the same code that you used to draw the Chinese coin in Chapter 2, except here the control points are a bit wonky.

Figure 10-7 depicts the amoeba outline with the vertex and control points visualized. The shape isn't perfectly round, but it is smooth with no discernible angles. For a smooth curve, the vertex and its two control points must form a straight line.

For a smooth curve, the vertex and its control points must form a straight line
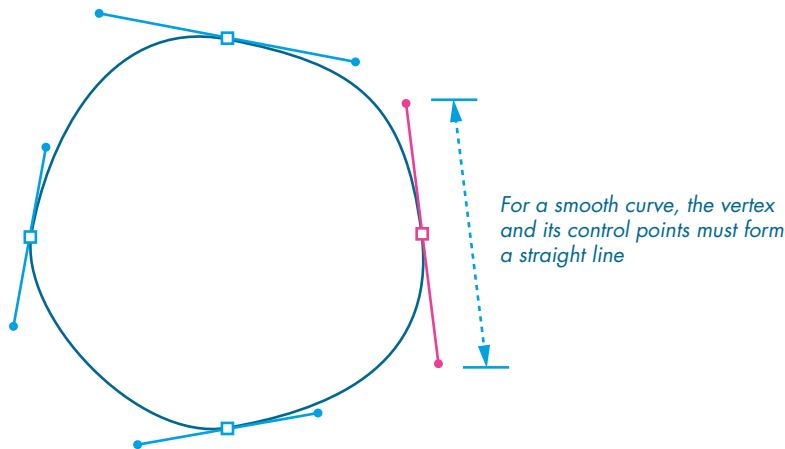
Figure 10-7: Drawing the amoeba with Bézier curves

To animate the wobble effect, you need to tweak the position of the control points for each frame. To avoid discernible angles and maintain the rounded appearance of the curves, you'll move your control points along circular paths. Figure 10-8 depicts (from left to right) two control points completing one rotation; each control point ends at the position it started, ready to repeat the motion seamlessly.

Notice that the opposite control point is always 180 degrees ahead of or behind its counterpart. As the control points near the vertex, the curve

grows tighter but remains rounded. The circular trajectories maintain the (virtual) straight line that runs from one control point to the other, through the vertex.
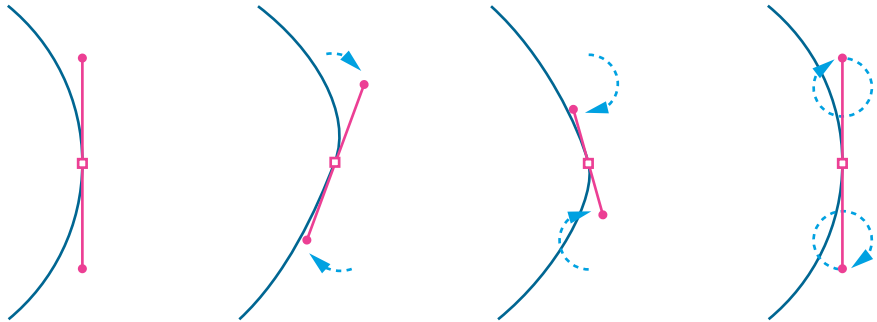
Figure 10-8: Moving the control-point coordinates along circular paths

To program this effect, add a `circlePoint()` method for calculating points along the perimeter of each circular path (this method is an adaption of the `circlePoint()` function you defined in Chapter 9):

```
class Amoeba(object):
    . . .
    def circlePoint(self, t, r):
        x = cos(t) * r
        y = sin(t) * r
        return [x, y]
    . . .
```

The `circlePoint()` method accepts two arguments, a theta (`t`) value and radius (`r`). The rules of function scope apply to methods too, so the variables `x` and `y` are local to the `circlePoint()` method.

You can call methods via the class instance—the `circlePoint()` method using `a1.circlePoint()`, for example. Of course, you'll need to include the two arguments (for `t` and `r`). You can also call a method from within its class by using a `self` prefix—for example, `self.circlePoint()`. In this way, you can call the `circlePoint()` method from within the `display()` function, using the returned values to draw wobbly amoeba.

Add a `circlePoint()` method call to the `display()` block, and replace the `circle()` function (for the cell membrane) with code for drawing a shape composed of `bezierVertex()` functions:

```
    . . .
    def display(self):
        . . .
        # cell membrane
        fill(0x880099FF)
        stroke('#FFFFFF')
        strokeWeight(3)
        r = self.d / 2.0
        cpl = r * 0.55
        cpx, cpy = self.circlePoint(frameCount/(r/2), r/8)
```

```
        xp, xm = self.x+cpx, self.x-cpx
        yp, ym = self.y+cpy, self.y-cpy
        beginShape()
        vertex(
          self.x, self.y-r  # top vertex
        )
        bezierVertex(
          xp+cpl, yp-r, xm+r, ym-cpl,
          self.x+r, self.y  # right vertex
        )
        bezierVertex(
          xp+r, yp+cpl, xm+cpl, ym+r,
          self.x, self.y+r  # bottom vertex
        )
        bezierVertex(
          xp-cpl, yp+r, xm-r, ym+cpl,
          self.x-r, self.y  # left vertex
        )
        bezierVertex(
          xp-r, yp-cpl, xm-cpl, ym-r,
          self.x, self.y-r  # (back to) top vertex
        )
        endShape()
. . .
```

The r variable represents the radius of the amoeba. The cpl value is the distance from each control point to its vertex; recall that this is roughly 55 percent of the circle radius for perfectly round circles (see Chapter 2, Figure 2-22). The circlePoint() method calculates the coordinates for variables cpx and cpy by using a theta value based on the advancing frameCount; the frameCount is divided by half the amoeba radius, so that larger amoeba wobble more slowly than smaller ones. The second circlePoint() argument, for the radius of the circular path, is also proportional to the amoeba radius. The rest of the code uses the cpl, cpx, and cpy variables to plot the vertices and curves that compose the wobbly amoeba.

Run the sketch to confirm that you have a wobbling amoeba.

Modifying an Attribute by Using a Method

You can use a method to modify one or many attributes as an alternative to changing values directly via dot notation. Here's a brief example; there's no need to add this code to your sketch.

When you instantiate your a1 amoeba, your __init__() method randomly selects a nucleus fill from a predefined list of five colors. You can change this by assigning another value via a1.nucleus['fill']. Alternatively, you might define a new method to do this for you:

```
class Amoeba(object):
    . . .
    def styleNucleus(self, fill):
        self.nucleus['fill'] = fill
    . . .
```

The `styleNulceus()` definition includes a parameter for a fill value. After you've instantiated amoeba a1, you can set the nucleus fill to black by using `a1.styleNucleus('#000000')` instead of `a1.nucleus['fill'] = '#000000'`. This might not seem very useful, but consider that you could add additional arguments for the nucleus dictionary's x, y, and d values to change them all at once. You might even add additional logic, like an `if` statement to check the size of a diameter value before applying it:

```python
def styleNucleus(self, fill, diameter):
    self.nucleus['fill'] = fill

    if diameter > self.d/4 and diameter < self.d/2.5:
        self.nucleus['d'] = diameter
```

The `styleNucleus()` definition now includes an additional parameter for the nucleus diameter. But the new diameter value applies only if it's appropriately sized. The `if` statement will ensure that the method ignores any value too small or too large so that you don't end up with a tiny nucleus or an over-size one that extends beyond the cell membrane.

Before moving on, here's a brief recap of where you're at in your amoeba simulation. You've defined an `Amoeba` class, complete with attributes to vary the appearance of each instance. You created a single amoeba, a1, but you'll add other instances soon. You defined an `__init__()` method to initialize the attributes. Additionally, you defined a `display()` method to draw the amoeba that calls another method, `circlePoint()`, to make the cell membrane wobble. Later, you'll make your amoebas move about the display window. First, though, you'll split your *microscopic* sketch into two files.

ABSTRACTION

This is a good point to discuss *abstraction*, the process of reducing something complex to a simpler form that provides what you really need to accomplish a task. For example, if you're designing a road map, you wouldn't include every real-world detail—just the drivable roads, bodies of water, and labels for major landmarks. In this way, a road map presents an abstracted version of a satellite image to assist navigation better.

To use another car analogy, you don't need to be a mechanic in order to drive. As long as you can operate a gear lever, steering wheel, and pedals, you can drive a car (never mind how well). Those instruments present an abstraction of your car's inner workings, providing an intuitive interface to control the transmission, steering system, and engine.

In Python, you use abstraction on many levels. For example, you call the `print()` function to display things in the console. The details of how Python makes this happen are irrelevant to you; this function represents a complex set of instructions abstracted down to a single `print()` call.

(continued)

In object-oriented programming, you design abstractions, deciding which details to hide and which to expose by way of attributes and methods. For example, in Python, a car object is an abstract representation of a real-world car using code. It's simplified, because you don't need to model each bolt, gear, and electrical wire to animate a vehicle moving about the screen. Moreover, the `Car` class will reduce a complex set of Python instructions—for appearance, movement, and so on—to a selection of intuitive methods, like `shiftGear()`, `steer()`, and `accelerate()`.

As a programmer, you must decide how you apply abstraction in your programs. This includes how you model objects in Python. The best approach is not always clear, and often there's no right or wrong way. Keep in mind, though, that good abstraction should make your code more clean, clear, modular, and maintainable.

Splitting Your Python Code into Multiple Files

In this book, you've worked through a series of relatively small programming tasks. Handling each sketch in a single file has been manageable enough, but your line counts will increase as you begin to work on more complex programs. You might squeeze a *Tetris* game into several hundred lines of Processing code, but the open source Minecraft-like game *Minetest* is almost 600,000 lines of (mostly) C++ code, and Windows XP comprises about 45 million lines of source code!

Programming languages have various mechanisms for structuring projects across multiple files. In Python, you can import code from files. Each Python file you import is referred to as a *module*. In this section, you'll create a separate amoeba module for your `Amoeba` class.

You'll need to consider the most sensible ways to divide any program into modules. For example, you might group a collection of related functions into a single module. Sometimes it's useful to add variables to a dedicated *configuration* module, providing a single location to set program-wide values. Grouping one or many related classes in a module is another great way to organize your code.

In the Processing editor, each tab represents a module. Create a new tab/module by using the arrow to the right of your *microscopic* tab, highlighted in magenta in Figure 10-9. From the menu that appears, select **New Tab**; name the new file *amoeba*.

Figure 10-9: Click the arrow tab, highlighted in magenta, for various tab operations.

This new file/module is created in the *microscopic* folder, alongside your main sketch file. Processing adds *.py* to the *amoeba* filename, the standard file extension for Python modules. The *amoeba.py* module should now appear as a tab alongside the microscopic one.

You can switch between your main sketch and modules by using the tabs. Switch to the *microscopic* tab and select all the code for your Amoeba class, cut it, and then switch to the *amoeba.py* tab and paste the code there (Figure 10-10).


```python
class Amoeba(object):

    def __init__(self, x, y, diameter):
        print('amoeba initialized')
        self.x = x
        self.y = y
```
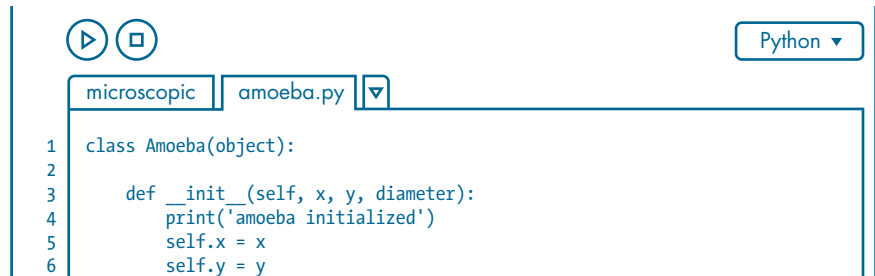
Figure 10-10: The amoeba.py *tab contains the code for your Amoeba class.*

Now switch back to the *microscopic* tab. What's left is everything from `a1 = Amoeba(400, 200, 100)` down.

To import modules, use the `import` keyword. Your `import` line must precede any code that instantiates an amoeba. Typically, `import` lines go at the top of files to avoid getting this sequence wrong. Here's the complete code for your *microscopic* tab:

```python
from amoeba import Amoeba

a1 = Amoeba(400, 200, 100)

def setup():
    size(800, 400)
    frameRate(120)

def draw():
    background('#004477')
    a1.display()
```

The `from` keyword instructs Python to open the *amoeba* module. The module takes its name from the filename, *amoeba.py*, but omits the *.py* extension. This is followed by `import` to specify the class(es) you want to import—in this case, `Amoeba`. This syntax allows you to be selective about which classes you import from modules that contain several class definitions. You can now use the `Amoeba` class as if it were defined in the *microscopic* tab.

Run the sketch. It should run as usual and display a single wobbling amoeba in the center of the display window.

You can use modules to share code among projects. For example, you can copy your amoeba module into any Processing project folder. Then, you simply import it to start creating amoebas. You can also store a collection of modules in a folder-type structure known as a *library* or *package.*

This modular system makes programming more efficient. In addition to reducing the line count of the main sketch, you conceal the inner workings of each module, leaving the programmer to focus on higher-level logic. For example, if you document your *amoeba* module, providing guidelines to instantiate amoebas and work the methods, any programmer can import and use it—creating amoebas without ever viewing the *amoeba.py* code. Additionally, modules make it easier for another programmer to browse your project code and understand your program because it's divided into named files.

Your a1 amoeba remains in a fixed position, wobbling as time passes. The next step is to get it moving about the display window.

Programming Movement with Vectors

You'll program your amoeba movement by using vectors. These are not the vectors for scalable graphics, though, but Euclidean vectors. A *Euclidean vector* (also known as a *geometric* or *spatial vector*) represents a quantity that has both magnitude and direction. You'll use vectors to model forces that propel your amoeba.

In Figure 10-11, the amoeba moves from position A to B; it's propelled a total distance of 4 units. This distance represents a magnitude; a *magnitude* describes how powerful a force is. A force with a greater magnitude might thrust the same amoeba 20 units. Here's the thing, though—the magnitude gives no indication of the direction in which the force is applied; you just know, from what you can glean visually, that the movement is 4 units to the right.

Figure 10-11: A magnitude of 4 units

A magnitude is a *scalar* value. It's a single quantity you can describe by using a single value, like a floating-point number or integer. For instance, the numbers 4, 1.5, 42, and one million are all scalar.

A *vector* is described by multiple scalars. In other words, it can hold multiple floating-point or integer values. Figure 10-12 presents a vector labeled *v* as a line with an arrowhead at one end. The length of *v* is its magnitude; the slope and arrowhead indicate its specific direction.
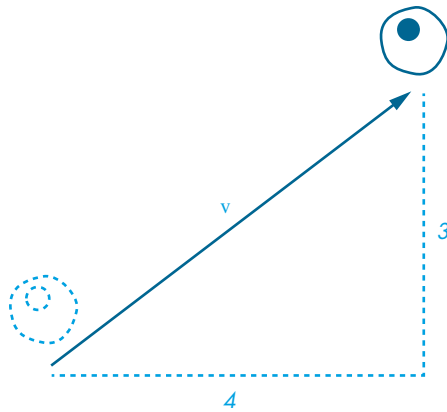
Figure 10-12: The vector v *extends 4 units right and 3 units up.*

Each vector has an x and y component, so you can express this vector as **v** = (4, 3). It describes a force to move the amoeba to a new location 4 units to the right and 3 units up from its previous location. You denote vectors in boldface type, but it's also common to draw a small arrow above the **v** in situations where bold is impractical (for example, for handwritten formulas).

The horizontal and vertical measurement lines in Figure 10-12 form a right triangle with *v* as its hypotenuse. From this triangle, you can calculate the magnitude of the vector by using the *Pythagorean theorem*. The theorem states that the square of the hypotenuse is equal to the sum of the squares of the other two sides.

If you add 4 squared (the adjacent side) to 3 squared (the opposite side), you get 25, the length of the hypotenuse squared. The square root of 25 is 5, the length of the hypotenuse and the magnitude of *v*. But you don't need to worry about performing such calculations. Processing provides a built-in PVector class especially for working with vectors that includes, among other methods, a mag() for calculating magnitude.

You'll adapt your amoeba sketch to work with the PVector class. While showing how to make your amoeba move with vectors, I'll also outline how the various PVector methods work, revealing what's happening on a mathematical level.

The PVector Class

PVector is a built-in Processing class for working with Euclidean vectors. You can use it anywhere in your sketch—no import line required. PVector can handle two- and three-dimensional vectors, but we'll stick to the 2D variety here.

To create a new 2D vector, the PVector() class requires an x and y argument. For example, this line defines the vector depicted previously in Figure 10-12:

```
v = PVector(4, 3)
```

The v instance is a new vector that extends 4 units across and 3 units up. You should, however, switch the 3 to -3 to match Processing's coordinate system (where the y values decrease as you move up).

A vector can point in any direction, negative or positive, but the magnitude is always a positive value. Use the mag() method to calculate the magnitude of any PVector instance; for example:

```
magnitude = v.mag()
print(magnitude)  # displays 5.0
```

You know that the mag() method must invoke prewritten code based on the Pythagorean theorem. It returns a floating-point value of 5.0, confirming our calculations from the previous section.

Moving an Amoeba with PVector

You'll create a PVector instance to animate amoeba a1 moving across the display window. In Chapter 6, you programmed something similar—a DVD screensaver—as you instructed Processing to move a DVD logo a set number of pixels horizontally and vertically in each frame for smooth, diagonal movements. The approach is similar here, but you'll use the PVector class instead. You'll find that the vector-based approach is more efficient for simulating movement and forces.

Switch to the *amoeba.py* tab and add a new propulsion vector to the __init__() method:

```
class Amoeba(object):

    def __init__(self, x, y, diameter, xspeed, yspeed):
        . . .
        self.propulsion = PVector(xspeed, yspeed)
```

The propulsion vector is initialized using two additional arguments for xspeed and yspeed that'll determine how many pixels your amoeba is propelled horizontally and vertically in each frame. In comparison to the DVD screensaver task, here you're combining the xspeed and yspeed variables into a single vector named propulsion.

Now switch to the *microscopic* tab. Use a fourth and fifth Amoeba() argument to set the x and y components of the propulsion vector to 3 and -1, respectively. Use the draw() function to increment your amoeba's x and y attributes by those values:

```
. . .
a1 = Amoeba(400, 200, 100, 3, -1)
. . .
def draw():
    background('#004477')
    a1.x += a1.propulsion.x
    a1.y += a1.propulsion.y
    a1.display()
```

Each frame, amoeba a1's x value increases by 3 pixels; at the same time, its y value decreases by 1. In the default Processing coordinate system, reducing y moves the amoeba up. If you run the sketch, the amoeba should move (quite rapidly) along a diagonal trajectory, starting in the center of the display window and soon exiting just below the upper right corner.

You can also use a `PVector` instance to store your amoeba's x- and y-coordinates. In fact, you can use `PVector` to store any x-y coordinate pair; after all, it's an object used to store two (or three) numbers, which also includes a bunch of handy methods for performing vector operations. Switch to the *amoeba.py* tab; replace the `self.x` and `self.y` attributes with a new vector named `self.location`:

```python
class Amoeba(object):

    def __init__(self, x, y, diameter):
        print('amoeba initialized')
        self.location = PVector(x, y)
        . . .
```

The amoeba's location is now a `PVector` instance too, albeit one that describes a point in the display window rather than a velocity or force. But you can't rerun the sketch yet. First, you need to update the rest of the *amoeba.py* file to work with the new location attribute.

Your `Amoeba` class has multiple references to `self.x` and `self.y`, and you'll need to ensure that you replace them all with `self.location.x` and `self.location.y`, respectively. The easiest way to do this is by using a find-and-replace operation. From the Processing menu bar, select **Edit ▸ Find** to access the **Find** tool (Figure 10-13). Enter **self.x** into the **Find** field, and **self.location.x** into the **Replace with** field. Click the **Replace All** button to apply the changes. The checkbox settings shouldn't make any difference here. Once you're done, do the same for `self.y`, replacing it with **self.location.y**.
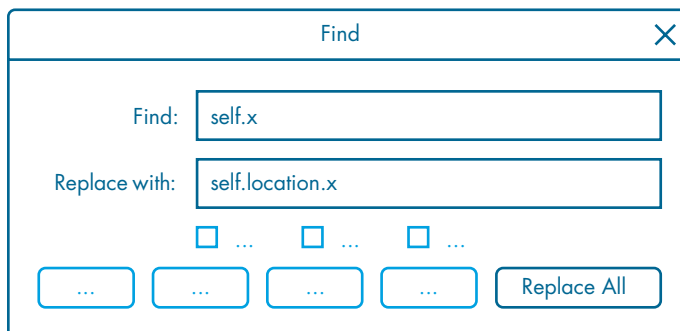
Figure 10-13: The Processing Find (and Replace) tool

Now, change `a1.x` and `a1.y` in your *microscopic* tab to `a1.location.x` and `a1.location.y`, respectively:

```
. . .
a1.location.x += a1.propulsion.x
a1.location.y += a1.propulsion.y
. . .
```

You add the x components on one line and the y components on another. However, there's a more efficient way to do this, using `PVector` addition.

Adding Vectors

The + operator is used to add floating-point numbers or integers. Additionally, it serves as a concatenation operator for string operands. The `PVector` class is programmed to work with the + operator too. You can add one `PVector` instance to another to get a vector that's the sum of the two. By extension, += works as an augmented assignment operator, stating that the vector operand to the left of the operator is equal to itself plus the right operand.

Replace your `a1.x += propulsion.x` and `a1.y += propulsion.y` lines with a single line to add the propulsion and location, adding `PVector` instances rather than individual components:

```
. . .
def draw():
    background('#004477')
    a1.location += a1.propulsion
    a1.display()
```

With each call of the `draw()` function (every frame), the amoeba location is incremented by the propulsion vector. If you run the sketch, the amoeba moves along the same trajectory as before, 3 pixels across and 1 up each frame, exiting just below the upper right corner of the display window.

Let's add a new force to the simulation. You'll model a current flowing diagonally across the display window; it assists the amoeba's prevailing motion, flowing toward northeast. As Wikipedia (*https://en.wikipedia.org/wiki/Current_(fluid)*) defines it, "A current in a fluid is the magnitude and direction of flow within that fluid." Evidently, this is something to model using a vector.

Add a new `PVector` named `current` to your *microscopic* tab. Add that vector to your location each frame by using the `draw()` function:

```
. . .
current = PVector(1, -2)
. . .
def draw():
    background('#004477')
    a1.location += a1.propulsion
    a1.location += current
    a1.display()
```

The propulsion vector is angled at roughly 18 degrees, pushing more rightward than upward. The current vector is angled at approximately 63 degrees, pushing more upward than rightward (Figure 10-14). This combination makes the amoeba move faster, at an angle somewhere between the two vectors (~36 degrees). If you run the sketch, the amoeba should exit the top edge of the display window (before, it exited at the right edge).
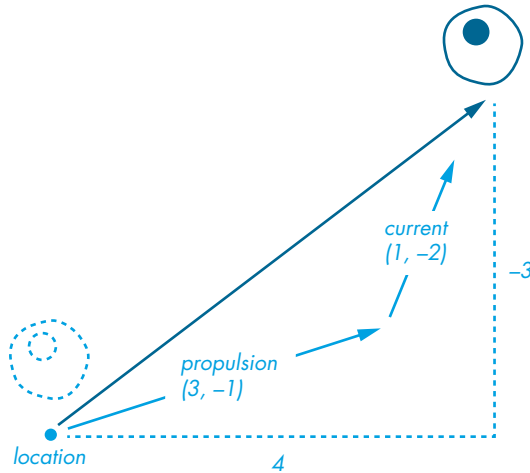
Figure 10-14: The amoeba moves a total of 4 pixels across and 3 up each frame.

Vector addition works by adding the x component of one vector to the x component of another, and likewise for the y components. In this case, adding the x components (3 + 1) equals 4, and adding the y components (−1 + −2) equals −3 . Regardless of the order in which you add vectors, the result is always the same. For example, (3, −1) + (1, −2) is the same as (1, −2) + (3, −1), and the resultant vector is (4, −3) in both instances. This makes vector addition a *commutative* operation, because changing the order of your operands doesn't change the result.

You can experiment with different current values to see what happens. A current vector of (−3, 1) cancels out the propulsion vector exactly, and the amoeba won't move from the center of the display window. A current vector of (−3.5, 1) will overpower the propulsion's x component and exactly match the y component, moving the amoeba slowly and directly leftward.

The neat thing about this system is that you can add as many forces to the object's location as you like. For instance, you might include a vector for wind, one for gravity, and so on.

Subtracting Vectors

In mathematics, the result of a subtraction operation is called the *difference*. For example, when you subtract 4 from 6, you're left with a difference of 2. Likewise, when you subtract one vector from another, the resultant vector is the difference between the two.

You can picture vector subtraction like this: begin by placing the two vectors tail to tail; between the head of each vector, draw a line; this new line is the difference vector. In Figure 10-15, you subtract *b* from *a*; the difference (dark blue vector *c*) is (–2, –1).
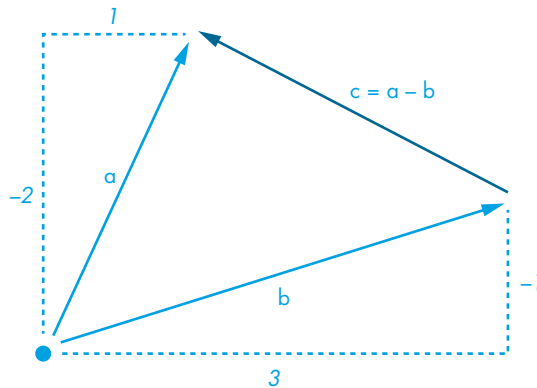
Figure 10-15: Vector c is equal to (–2, –1).

The process of vector subtraction is similar to vector addition, but rather than adding the x (with x) and y (with y) components of each vector, you're subtracting them. Note, however, that subtraction is *noncommutative*. That means, changing the order of the operands changes the result. For example, if you subtract *a* from *b*, you get (2, 1) instead of (–2, –1). This makes vector *c* point the opposite way, switching its head and tail.

You can subtract `PVector` instances by using the - operator. Here's an example:

```
print(current - a1.propulsion)
```

If your current vector is equal to (1, –2), this will print [`-2.0, -1.0, 0.0`] to the console. Processing prints a `PVector` instance as a list of three floating-point values, which represent the vector's x, y, and z components, respectively. The z value is always a 0, unless you're working with three-dimensional vectors.

You've added a propulsion and current vector to the amoeba's location to get it moving across the display window. You'll now apply what you've learned about vector subtraction to get the amoeba moving toward your mouse pointer. You'll create a new `PVector` instance called `pointer` to store the x-y coordinates of your mouse pointer. You'll subtract `location` (which holds the amoeba's x-y coordinates) from `pointer` to find the difference vector (Figure 10-16), which you'll use to redirect the amoeba.
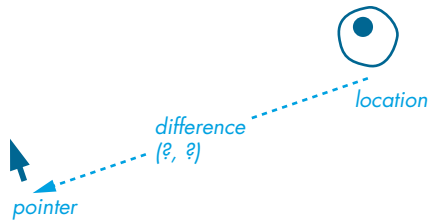
Figure 10-16: The difference vector is equal to pointer − location.

Ensure that your current vector is set to (1, −2). Add a new `PVector` named pointer and a difference variable that's equal to the pointer minus the amoeba location (the difference vector depicted in Figure 10-16).

```
. . .
current = PVector(1, -2)
. . .
def draw():
    background('#004477')
    pointer = PVector(mouseX, mouseY)
    difference = pointer - a1.location
    a1.location += difference
    . . .
```

The `mouseX` and `mouseY` are Processing system variables that hold the x- and y-coordinates of your mouse pointer. Note, however, that Processing can begin tracking the mouse position only after you move the pointer in front of the display window; until that time, `mouseX` and `mouseY` both return a default value of 0.

If you run the sketch, the amoeba will attach to the mouse pointer. This happens because the amoeba reaches the pointer position in a single "leap." Instead, you want the amoeba to "swim" toward the pointer, advancing in small increments over multiple frames.

Limiting Vector Magnitude

The `PVector` class provides the `limit()` method to limit the magnitude of any vector, which does not affect the direction. It requires a scalar (integer or floating-point) argument that represents a maximum magnitude.

You'll use the difference vector to steer the amoeba toward the mouse pointer by adding it to the propulsion vector. You'll limit the propulsion vector to a magnitude of 3 (Figure 10-17), enough to overpower the current marginally (which has a magnitude of 2.24) when the amoeba is swimming directly into it.
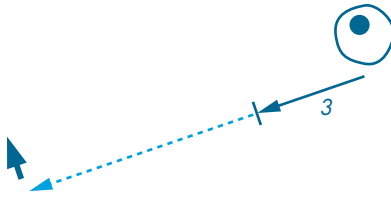
Figure 10-17: The propulsion vector's magnitude is limited to 3.

Make the following insertions/changes to the `draw()` function to steer and propel the amoeba toward the mouse pointer:

```
. . .
def draw():
    . . .
❶  #a1.location += difference
❷  a1.propulsion += difference.limit(0.03)
❸  a1.location += a1.propulsion.limit(3)
    a1.location += current
    a1.display()
```

First, comment out or delete the existing `a1.location += difference` line ❶. The `limit()` method restricts the `difference` vector to a magnitude of 0.03 ❷. This tiny value is added to the propulsion vector each frame—the effect rapidly accumulating—steering the amoeba progressively toward the mouse pointer. But even when the amoeba is heading directly at the pointer, the propulsion vector's magnitude will not exceed 3 ❸.

Run the sketch and position your mouse pointer over the display window somewhere near the lower left corner. The amoeba will have drifted out of view. But wait for a while, and it'll slowly make its way toward the corner; when it reaches the pointer, it will overshoot it slightly, then turn around and overshoot it on the way back. It continues to overshoot the pointer, because it's trying to reach its target as quickly as possible. Now move your pointer to the lower right corner. Assisted by the current, the amoeba is quick to reach the opposite side of the screen, but its higher velocity leads it to overshoot the target dramatically.

Soon, you'll add multiple amoebas to the simulation. To prepare them for moving at different speeds, add an attribute for maximum propulsion to the `Amoeba` class:

```
class Amoeba(object):

    def __init__(self, x, y, diameter, xspeed, yspeed):
        . . .
        self.maxpropulsion = self.propulsion.mag()
```

This attribute will limit the magnitude/power of the amoeba's propulsion vector based on the `xspeed` and `yspeed` arguments you provide. Adapt the code

in your *microscopic* tab to work with the `maxpropulsion` attribute, switching out the arguments of both `limit()` methods. Additionally, adjust the values for the `xspeed`, `yspeed`, and the current vector, reducing them by a factor of 10:

```
. . .
a1 = Amoeba(400, 200, 100, 0.3, -0.1)
current = PVector(0.1, -0.2)
. . .
def draw():
    . . .
    a1.propulsion += difference.limit(a1.maxpropulsion/100)
    a1.location += a1.propulsion.limit(a1.maxpropulsion)
    . . .
```

The reduced propulsion and current values slow down the simulation, so the amoeba movement is more steady and controlled. The amoeba won't wildly overshoot its target anymore, but it still makes small orbits around the pointer. The limit for the difference vector is now proportional to the amoeba's maximum propulsion, so a faster amoeba has some extra steering power to handle its higher velocity.

Performing Other Vector Operations

There's more to vectors and the `PVector` class, but that's all I cover in this book. Consider what you've learned as an elementary introduction to the topic. The `PVector` class can additionally handle vector multiplication, division, normalization, 3D vectors, and more. Vectors are useful for programming anything that requires physics, like video games, and you're likely to reencounter them in your creative coding adventures.

Adding Many Amoebas to the Simulation

You have a working amoeba module, but you're still dealing with a single amoeba instance, `a1`, so the next step is to create a colony. You can create as many instances as you like from a single class. In this section, you'll spawn eight amoebas in the same display window by using the `Amoeba` class. Each amoeba will vary in size, and you'll start them at different x-y coordinates. Recall that each amoeba instance includes a dictionary of randomized nucleus values, so the nuclei will vary too.

One (rather manual) approach for adding amoebas is to define additional instances with personalized variable names, with explicitly differentiated parameters. Consider these three new amoeba:

```
a1 = Amoeba(400, 200, 100, 0.3, -0.1)
sam = Amoeba(643, 105, 56, 0.4, -0.4)
bob = Amoeba(295, 341, 108, -0.3, -0.1)
lee = Amoeba(97, 182, 198, -0.1, 0.2)
. . .
```

You can keep adding amoebas in this manner, but the approach has its downsides. For one, you need to remember to call every `display()` method in the body of the `draw()` function to render each amoeba:

```
def draw():
    . . .
    a1.display()
    sam.display()
    bob.display()
    lee.display()
    . . .
```

This will display sam, bob, and lee standing still; to get those amoebas moving, the `draw()` function requires even more code. That isn't especially efficient if you're dealing with 5 or so amoebas, never mind 100.

Personalized amoeba names are cute and all, but not important for this program. Instead, you'll store the amoebas in a list. You can conveniently use a loop to generate a list of as many amoebas as you like. Then you can call each amoeba's `display()` method (along with the code to move it) by using another loop.

Replace the a1 line at the top of your *microscopic* code with an empty amoebas list and a loop to populate it:

```
from amoeba import Amoeba

amoebas = []

for i in range(8):
    diameter = random(50, 200)
    speed = 1000 / (diameter * 50)
    x, y = random(800), random(400)
    amoebas.append(Amoeba(x, y, diameter, speed, speed))
. . .
```

With each iteration of the `for` loop, Python creates a new `Amoeba()` instance. The `Amoeba()` arguments are randomized to vary the x-coordinate, y-coordinate, and diameter of each instance. The speed value is based on the diameter—so bigger amoebas move slower (recall that the propulsion and maxpropulsion attribute is derived from the xspeed and yspeed arguments). The append() method adds the new amoeba instance to the amoebas list. The amoebas don't have names like sam, bob, and lee, but you can address them by index as amoebas[0], amoebas[1], and so forth.

You must add a `for` loop to the `draw()` function to render the full list of amoebas. Here's your amended code:

```
. . .
def draw():
    background('#004477')
    pointer = PVector(mouseX, mouseY)
```

```
for a in amoebas:
    difference = pointer - a.location
    a.propulsion += difference.limit(a.maxpropulsion/100)
    a.location += a.propulsion.limit(a.maxpropulsion)
    a.location += current
    a.display()
```

The `for` loop iterates the entire `amoebas` list. For each amoeba, it calculates an updated location, and then renders that amoeba by using its `display()` method.

The larger, slower amoebas might drift out of the display window, overwhelmed by the current, never to be seen again. To avoid this problem, add code for *wraparound* edges—so that if an amoeba exits the display window, it reappears on the opposite side, maintaining its speed and trajectory:

```
. . .
for a in amoebas:
    . . .
    r = a.d / 2

    if a.location.x - r > width:
        a.location.x = 0 - r
    if a.location.x + r < 0:
        a.location.x = width + r
    if a.location.y - r > height:
        a.location.y = 0 - r
    if a.location.y + r < 0:
        a.location.y = height + r
```

The four `if` statements check each edge of the display window. It's necessary to incorporate the radius (variable r) in the conditions to ensure that the amoeba has fully left the display window before it reappears on the opposite side. Likewise, each corresponding destination is offset by r to prevent the amoeba from reappearing halfway over the opposite edge. You can set r to 0 if you'd like to see what happens otherwise.

Each time you run the sketch, you get a different selection of amoebas. They all swarm toward your mouse pointer (although the current overpowers some of the large, slow ones), overlapping one another in the process. Figure 10-18 shows an example with eight amoebas.

To add or remove amoebas, you can adjust the argument in the `range()` function of your first loop, and the loop in the `draw()` function will adapt dynamically. If your computer seems to be struggling, you can reduce the number of amoebas.
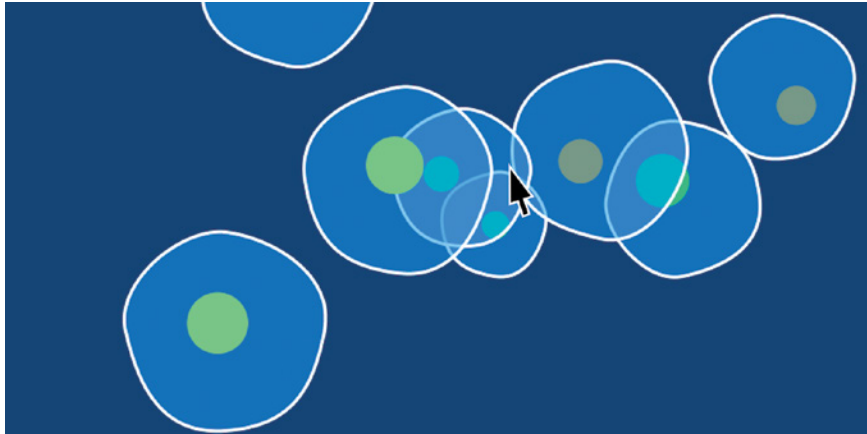
Figure 10-18: A display window with eight amoebas moving toward the mouse pointer

Challenge #10: Collision Detection

The amoebas can overlap one another. To prevent this from happening, you must first detect where overlaps occur. From there, you can apply vector forces to push any colliding pairs apart.

The amoebas are roughly circular, so a *circle-circle collision detection* algorithm will work nicely here. To understand how circle-circle collision detection works, refer to Figure 10-19. The pair of circles on the left have not collided; on the right is a colliding pair. For the non-colliding circles, the distance between the centers of each circle is greater than the sum of the two radii (r1 and r2). Conversely, where the circles have collided, the distance is less than the sum of the two radii.
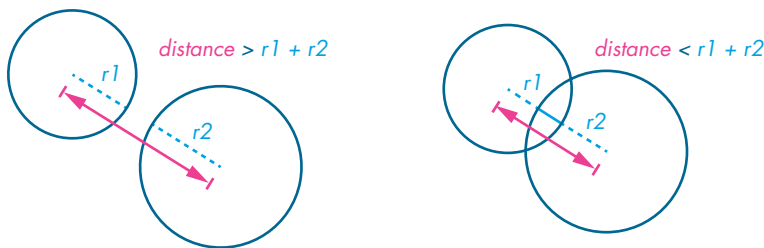
Figure 10-19: Circle-circle collision detection

To test for collisions in Processing, you'll need to check each amoeba against every other amoeba in the amoebas list. For this purpose, add another for loop within the a in amoebas loop:

```
. . .
for a in amoebas:
    . . .
```

```
        for b in amoebas:

            if a is b:
                continue

            # your solution goes here
```

You don't want to check whether an amoeba is colliding with itself. At the top of the loop, there's an `if a is b` test. The `is` operator compares the objects on either side of itself to determine whether they point to the same instance; if `a` is the same instance as `b`, this will evaluate as `True`. The `continue` line terminates the current iteration of the loop to start at the beginning of the next, so your "solution" code is skipped.

Think about how you can use the distance vectors shown in Figure 10-19 to push apart colliding amoebas. Can you add (or subtract) a fraction of the distance vector to push an amoeba in the opposite direction to the one it has collided with?

If you need help, you can access the solution at *https://github.com/ tabreturn/processing.py-book/tree/master/chapter-10-object-oriented_programming _and_pvector/.*

Summary

In this chapter, you learned how to use object-oriented programming to model real-world objects in Python. You defined a new `Amoeba` class, to which you added attributes and methods. A class serves as an object template, from which you can create countless instances. Grouping related variables (attributes) and functions (methods) into classes can help you structure code more efficiently. This is especially effective for programming larger, more complex projects.

You also learned how to separate classes (and other code) into different Python files, called modules, and how to use those modules to share code between projects or as reusable components among files in the same project. Remember that modules reduce the line count of the main sketch, allowing you to focus on higher-level logic.

This chapter also introduced Processing's built-in `PVector` class for dealing with Euclidean vectors. A Euclidean vector describes a quantity that has both magnitude and direction, but you can also use a vector to store something's location (as an x-y coordinate). In this chapter, you used vectors to simulate forces and control the positions of various objects in the display window.

In the next chapter, you'll learn how to handle mouse and keyboard interaction in Processing. I've already touched on the `mouseX` and `mouseY` system variables in this chapter. However, you can do much more with capturing mouse clicks and keypresses, unlocking exciting ways to interact with your Processing sketches.

11

MOUSE AND KEYBOARD INTERACTION

In this chapter, you'll learn how to program interactive sketches that respond to mouse and keyboard input. You can combine those input devices in interesting and useful ways. For example, many computer games use a combination of keys for player movement and the mouse for aiming. Here, you'll program sketches that use the mouse to paint as well as to select items from a tool palette. You'll also add shortcut keys to activate tools by using the keyboard.

This chapter introduces system variables you can use to monitor mouse clicks and keypresses. You'll also learn about *event functions* that execute whenever a particular type of mouse or keyboard event occurs.

The first task you'll complete is a simple doodling program. The second is a more elaborate paint app that includes a tool palette for selecting colors

and brushes. When you're programming interactive sketches, you need to be able to change the contents of the display window in response to user input so both programs are set up as animated sketches.

Mouse Interaction

You can use mouse input to perform point-and-click operations. You can also program gesture-type motions that combine mouse movements and clicks, like a drag-and-drop or pan. Most mice include three buttons—a left button, a right button, and a clickable scroll wheel that doubles as a center button.

Mouse Variables

Processing's mouseX and mouseY variables hold the horizontal and vertical position of your mouse pointer in the display window. Processing also provides the system variables pmouseX and pmouseY that contain the mouse coordinates of the previous frame. There's also a mousePressed variable that's set to True whenever a mouse button is held down.

The first task in this chapter focuses on Processing's mouse variables to monitor the position of the mouse pointer and detect when mouse buttons are pressed. You'll program a simple sketch for making *scratch art*. A sheet of scratch art paper is covered with a rainbow blend of colors, then coated in a layer of black (Figure 11-1). The "scratcher" uses a plastic or wooden stylus to etch lines in the black surface, revealing the colors beneath.
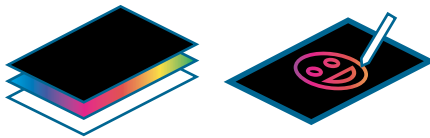
Figure 11-1: The layers that make up a sheet of scratch art paper (left), and doodling with a stylus (right)

You could buy scratch paper ready-made or make it yourself, but pixels are cheap and reusable!

Create a new file and save it as *scratch_art*. Add the following code to get your mouse drawing a trail of white circles:

```
def setup():
    size(800, 400)
    frameRate(20)
    background('#000000')
    stroke('#FFFFFF')

def draw():
    circle(mouseX, mouseY, 15)
```

Each frame, Processing draws a new circle positioned using the mouseX and mouseY values. Those coordinates are retrieved once each frame, and

the frame rate is relatively low (20 fps). The draw() block has no background() function, so every circle drawn persists until you close the display window. If you move the mouse slowly, the circles will form a solid white line; wherever you move the mouse quickly, you'll leave discernible gaps in the line (Figure 11-2). The upper left corner of the display window always contains a circle, because the first x-y coordinate pair for the mouse is equal to (0, 0).

Figure 11-2: The faster you move your mouse, the larger the gaps in the line.

You can increase the frame rate to fill in the line better, but you'll still get gaps if you move the mouse fast enough. To ensure a continuous line, replace the circle() function with code for drawing lines:

```
. . .
def draw():
    strokeWeight(15)
    line(mouseX, mouseY, pmouseX, pmouseY)
```

The strokeWeight() argument of 15 matches the circle diameter from before. The line() function draws a line between the mouse coordinates of the current frame and those of the previous frame.

Run the sketch. The first line() function will extend from the upper left corner (0, 0) to where your mouse pointer first enters the display window. In Figure 11-3, my mouse enters the display window from the left edge (I'm drawing the wave from left to right).

Figure 11-3: Drawing a continuous line by using a line() function for each frame

To turn the "brush" on and off, insert an `if` statement to activate the `line()` function while the left mouse button is being pressed:

```
. . .
def draw():
    strokeWeight(15)
    if mousePressed and mouseButton == LEFT:
        line(mouseX, mouseY, pmouseX, pmouseY)
```

The `mousePressed` variable holds a Boolean value that's set to `True` while any mouse button is pressed. The `mouseButton` variable is equal to `LEFT`, `RIGHT`, or `CENTER`, depending on which button is pressed, and `0` if it's none of them.

NOTE *The `mousePressed` variable will revert to `False` as soon as the button is released; the `mouseButton` variable, however, will retain its value until a different button is pressed (or until the mouse is moved).*

Run the sketch to confirm that Processing draws a white line while you're holding down the left-mouse button.

For the rainbow-color scratch art effect, insert code to base the stroke color on the position of the mouse pointer. The horizontal position will control hue, and the vertical position will control saturation:

```
. . .
def draw():
    colorMode(HSB, 360, 100, 100)
    h = mouseX * 360.0 / width
    s = mouseY * 100.0 / height
    b = 100
    stroke(h, s, b)
    . . .
```

In this example, you set the color mode to `HSB` (hue, saturation, brightness). The `h` variable is assigned a hue value between 0 and 360; the `s` variable is assigned a saturation value between 0 and 100. Both the `h` and `s` values are based on the mouse pointer's position relative to the width and height of the display window. The color's brightness value is always 100 percent.

Run the sketch to test out the finished scratch art program (Figure 11-4).

Figure 11-4: Doodling in the scratch art program

Now that you've seen Processing's mouse variables, in the next task, you'll learn about mouse event functions.

Mouse Events

Processing offers a selection of mouse event functions that run each time a particular mouse event takes place. Those functions are `mouseClicked()`, `mouseDragged()`, `mouseMoved()`, `mousePressed()`, `mouseReleased()`, and `mouseWheel()`. You add code to an event function block that executes when the event function is triggered. To illustrate how this works, I'll compare an example using a `mousePressed` variable with another that employs a `mousePressed()` event function.

The following code uses the `mousePressed` system variable to switch the background color from red to blue with the press of a mouse button:

```
def draw():
    background('#FF0000')  # red

    if mousePressed:
        background('#0000FF')  # blue
```

The background color is blue as long as the user is holding down a mouse button; otherwise, it's red. The next example uses a mouse event— a `mousePressed()` function—to perform a similar operation:

```
def draw():
    background('#FF0000')  # red

def mousePressed():
    background('#0000FF')  # blue
```

Each time a mouse button is pressed, the `mousePressed()` function executes the blue background line, and the display window will briefly flash blue (for a single frame). It immediately returns to red, regardless of how long you hold down the mouse button. This is because an event function executes just once per event; in other words, the background will not flash blue again until you release and then re-press the mouse button.

Creating a Paint App

For the next exercise, you'll program a basic painting app that will feature a tool palette for selecting color swatches and other options. You'll use the `mousePressed()`, `mouseReleased()`, and `mouseWheel()` functions.

The large dark blue area, to the right in Figure 11-5, is your canvas for drawing; the tool palette sits against the left edge. You hold down the left-mouse button to draw.
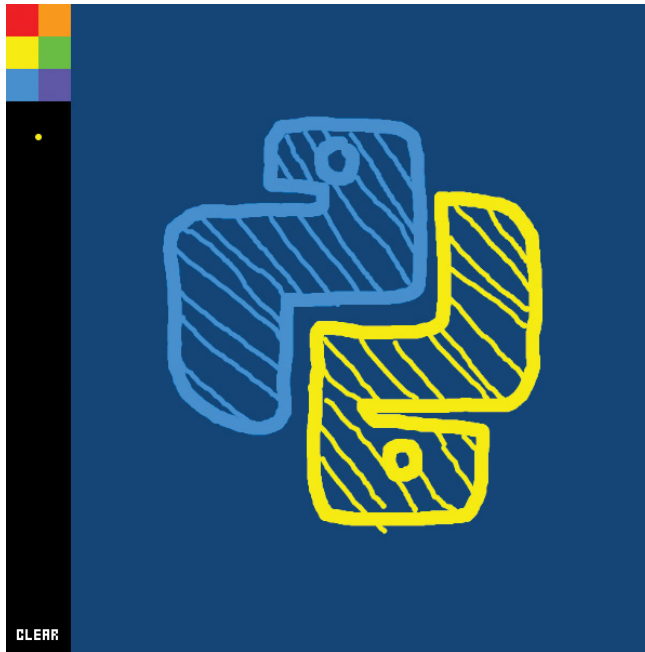
Figure 11-5: The paint app with a (badly drawn) Python logo

To begin, create a new sketch and save it as *paint_app*. You'll use the Ernest font, created by Marc André "Mieps" Misman, to label the buttons in your tool palette. Download this font from the book's GitHub web page:

1. Open your web browser and go to *https://github.com/tabreturn /processing.py-book/.*
2. Navigate to *chapter-11-mouse_and_keyboard_interaction.*
3. Download the *Ernest.ttf* file.
4. Create a new *data* subfolder within your sketch folder and place the *Ernest.ttf* file in it.

Add the following code to set up your sketch. This defines the display window size, background color, font, and global variables for the paint app:

```
def setup():
    size(600, 600)
    background('#004477')
    ernest = createFont('Ernest.ttf', 20)
    textFont(ernest)

swatches = ['#FF0000', '#FF9900', '#FFFF00',
            '#00FF00', '#0099FF', '#6633FF']
brushcolor = swatches[2]
brushshape = ROUND
brushsize = 3
painting = False
```

```
paintmode = 'free'
palette = 60
```

You'll use the global variables (`swatches`, `brushcolor`, and so on) to adjust and monitor the state of the brush. The default brush color is set to yellow. Later, you'll use the `palette` variable to set the width of the tool palette. You haven't added anything visual yet, so if you run the sketch, all you'll see is a plain blue display window.

Controlling the Draw Loop with loop() and noLoop() Functions

You'll control the `draw()` function behavior by using mouse events. While the left mouse button is pressed, the `draw()` function will loop; once it's released, the looping stops, which is a convenient way to control how the paint app works. Of course, the `draw()` function loops by default, so you'll need the `loop()` and `noLoop()` functions to take control of this behavior.

The `noLoop()` function stops Processing from continually executing the code in the `draw()` block. A `loop()` function reactivates the standard `draw()` function behavior, and a `redraw()` function is available if you need to execute the `draw()` code just once.

To start, add a `noLoop()` function to the `setup()` block, and a `draw()` function that prints the frame count:

```
def setup():
    . . .
    noLoop()
. . .
def draw():
    print(frameCount)
```

If you run the sketch, the console should display a single 1, confirming that the `draw()` function has run only once.

Now add code to the `draw()` function to make the mouse draw lines in the display window, and add two mouse events to start and stop the flow of paint:

```
. . .
def draw():
    print(frameCount)
    global painting, paintmode

❶  if paintmode == 'free':

❷      if painting:
            stroke(brushcolor)
            strokeCap(brushshape)
            strokeWeight(brushsize)
❸          line(mouseX, mouseY, pmouseX, pmouseY)

❹      elif frameCount > 1:
            painting = True
```

```
❺ def mousePressed():
        # start painting
        if mouseButton == LEFT:
            loop()

❻ def mouseReleased():
        # stop painting
        if mouseButton == LEFT:
            global painting
            painting = False
            noLoop()
```

Read over this code while simulating the process in your mind, paying careful attention to when `painting` is equal to `True` or `False`, and when the `draw()` function is running continuously. The sketch begins with the `painting` variable set to `False`; the `draw()` function isn't looping at this point. When you press the left mouse button ❺, the `loop()` function instructs Processing to resume looping the `draw()` function; when you release the button ❻, the `noLoop()` function halts the draw behavior again. The `paintmode` variable is set to `free` ❶ by default, so Python checks whether you're currently painting ❷. You'll add other paint modes later. If `painting` is equal to `True`, Processing draws a line between the mouse coordinates of the current frame and those of the previous frame ❸; if not, it checks that the frame count has passed 1 ❹ before setting the `painting` variable to `True`. The `if` ❷ and `elif` ❹ steps are necessary to avoid drawing straight lines between where you cease and resume painting (release the left button, move the mouse, and then press the button again), and the `frameCount > 1` stops Processing from drawing a line from the upper left corner to where you first begin painting. In Figure 11-6, the left screenshot depicts what happens if you omit those statements.
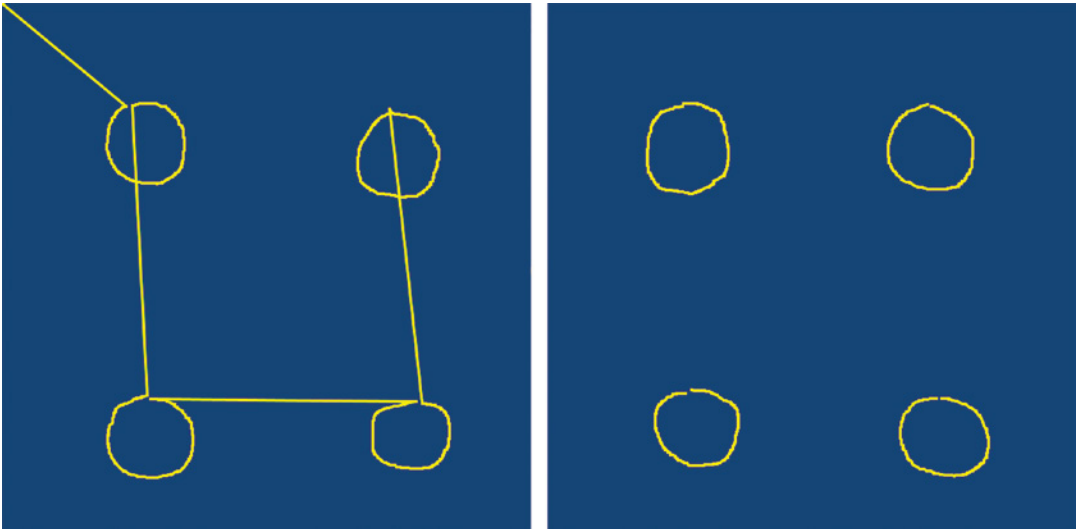
Figure 11-6: Processing drawing straight lines between the points where painting stops and starts (left), and your version of the program (right)

Run the sketch, and draw a few circles to test that the code is working. Watch the console and note that the frame count increases only while you're pressing the left mouse button.

Adding Selectable Color Swatches

The tool palette will include six color swatches that you can use to change the brush color. Add the following code to the bottom of the draw() block to render a black panel against the left edge of the display window, and within it, six color swatches based on the swatches list:

```
. . .
def draw():
    . . .
    # black panel
    noStroke()
    fill('#000000')
    rect(0, 0, palette, height)
    # color swatches
    for i, swatch in enumerate(swatches):
        sx = int(i%2) * palette/2
        sy = int(i/2) * palette/2
        fill(swatch)
        square(sx, sy, palette/2)
. . .
```

The for loop iterates the swatches list, drawing a grid of squares filled in different colors. Your program draws the panel (and swatch elements) after the brushstrokes to prevent unwanted strokes appearing over the palette as you select things.

If a user clicks a color swatch, you must assign that color to the brushcolor variable; add code for this to the mousePressed() function:

```
. . .
def mousePressed():
    . . .
    # swatch select
    if mouseButton == LEFT and mouseX < palette and mouseY < 90:
        global brushcolor
        brushcolor = get(mouseX, mouseY)
```

The if statement tests for a left-click and checks that the mouse pointer is positioned somewhere over the color swatches. The get() function returns the color for the pixel under the mouse pointer and assigns it to the brushcolor variable. You add a global line to overwrite the brushcolor variable in the global scope, the same variable that the draw() function uses to apply the stroke for the brush color.

Run the sketch. You can now select colors for painting (Figure 11-7).
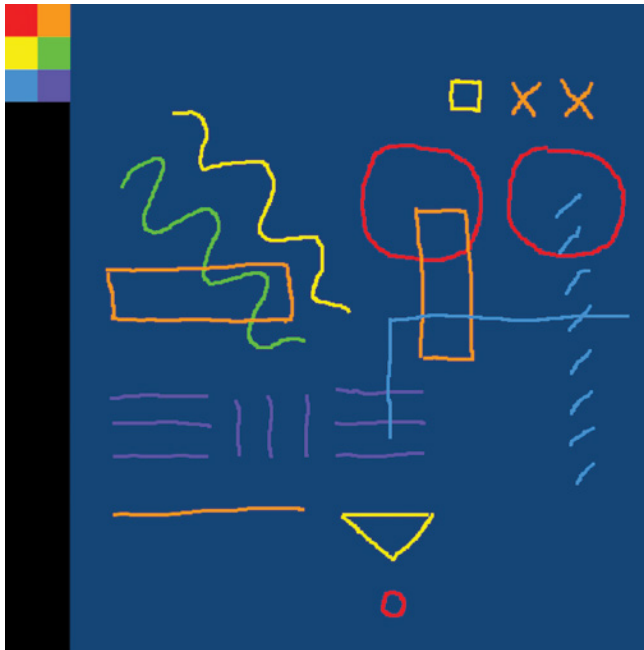
Figure 11-7: Click the swatches in the tool palette to change the brush's color.

Next, you'll add a feature for resizing the brush, mapping this to the scroll wheel.

Resizing the Brush by Using the Scroll Wheel

The `mouseWheel()` event function is used to execute code when the mouse wheel is moved. Additionally, you can use it to retrieve positive or negative values depending on the direction of the scroll wheel's rotation. The direction of rotation for positive/negative values, however, depends on your system's configuration. Your touch pad scroll should work for this too, usually with a two-finger drag.

Add a `mouseWheel()` function to the very bottom of your code:

```
. . .
def mouseWheel(e):
    print(e)
```

The `e` within the `mouseWheel()` function's parentheses serves as a variable to which the event details are assigned. You may name this whatever you like; programmers commonly use `e` or `event`.

Run the sketch, position your mouse pointer somewhere over the display window, and use the scroll wheel. The console should display something like the following:

```
<MouseEvent WHEEL@407,370 count:1 button:0>
```

From this output, you can establish that the type of mouse event is WHEEL. At the moment of the event, the horizontal mouse position is 407, and the vertical position is 370 (@407,370). The scroll direction is positive (count:1). Of course, your values will vary somewhat.

NOTE *You can also add this event variable to one of the other mouse functions—like mousePressed() or mouseReleased(). As an example, for mousePressed(e), the e might hold something like <MouseEvent PRESS@407,370 count:1 button:37>.*

Add code that uses the mouseWheel() function to adjust the brush size. This code will also display a brush preview below the swatches, which will reflect the active brush's color, size, and shape:

```
. . .
def draw():
    . . .
    # brush preview
    fill(brushcolor)
    if brushshape == ROUND:
        circle(palette/2, 123, brushsize)
❶ paintmode = 'free'

. . .

def mouseWheel(e):
    # resize the brush
    global brushsize, paintmode
❷ paintmode = 'select'
❸ brushsize += e.count
❹ if brushsize < 3:
        brushsize = 3
❺ if brushsize > 45:
        brushsize = 45
    redraw()
```

You don't want to paint while adjusting the brush size, so paintmode is switched to select ❷. The e.count is used to retrieve the negative/positive scroll value from the event variable, which is added to brushsize ❸. It's necessary, however, to include checks (if statements) to ensure that the new brush size remains within a reasonable range (between 3 ❹ and 45 ❺). Finally, the redraw() function runs the draw() function, just once, to update the brush preview and switch paintmode back to free ❶.

Run the sketch to confirm that you can resize the brush by using the scroll wheel, which updates the brush preview in the palette (Figure 11-8).

Figure 11-8: Painting with brushes of different sizes

There's one problem, though. When selecting swatches with a large brush, a blob of paint might extend into the dark blue canvas area (Figure 11-9).

Figure 11-9: Selecting a color swatch with a large brush

To solve this problem, add an `if` statement to the `draw()` function that disables painting while the mouse is over the palette. Use the `paintmode` variable to control this:

```
. . .
def draw():
    print(frameCount)
    global painting, paintmode

    if mouseX < palette:
        paintmode = 'select'
    . . .
```

Run the sketch to confirm that you can select color swatches with large brushes, without blobs encroaching on the canvas area.

Now that you understand how mouse events work, consult the online reference if you need a `mouseDragged()` or `mouseMoved()` function. You'll try out a `mouseClicked()` function in "Challenge #11: Adding Paint App Features" on page 252. If you'd like to change the mouse pointer from an arrow to something else, you can use the `cursor()` function. For example, you can add a `cursor(CROSS)` function to the `setup()` block for a crosshair.

Keyboard Interaction

Computers inherited their keyboard designs from typewriters. In the process, computer keyboards spawned various new keys, such as the arrow, escape, and function keys, and a number pad for more efficient numeric entry. They also have modifier keys (like ALT and CTRL) that you can use in conjunction with other keys to perform specific operations. For example, the Z, X, C, and V keys combine with CTRL or ⌘ to perform undo/cut/copy/paste operations.

NOTE *The typewriter's SHIFT key could be credited as the original modifier key, taking its name from how it physically* shifted *a substantial part of the typewriting mechanism into a position for typing uppercase letters.*

Keyboard interaction in Processing works similarly to mouse interaction, with system variables such as key and keyPressed, and event functions like keyPressed(), keyReleased(), and keyTyped().

Now, let's add keyboard shortcuts to the paint app for selecting colors.

Adding Keyboard Shortcuts to the Paint App

To program the shortcuts, you'll combine the key system variable and keyPressed() event function. The keyPressed() function runs once each time a key is pressed. However, holding down a key may cause repeated function calls. The operating system controls this repetitive behavior, and configurations can vary among users. Processing stores the value of the most recently used key in the key system variable.

Add a keyPressed() event function to the end of your code. For now, this will print the key value in the console:

```
. . .
def keyPressed():
    print(key)
```

Run the sketch and press different keys. Numbers, letters, and symbols display in the console as you might expect them to—you get a 1 when you press the 1 key, a q when you press the Q key, and so on. If CAPS LOCK is on, you get uppercase letters.

NOTE *Special keys, like the arrow and modifier keys, work a little differently. If you need to detect those, refer to the reference entry for* keyCode*.*

To select different color swatches, replace the print() function with code that uses the number keys 1 to 6:

```
. . .
def keyPressed():
    global brushcolor, paintmode
    paintmode = 'select'
    # color swatch shortcuts
```

```
        if str(key).isdigit():
            k = int(key) - 1
            if k < len(swatches):
                brushcolor = swatches[k]
                redraw()
```

Python's isdigit() method returns True if all of the characters in a string are digits. This works only on characters/strings and will handle most key values fine, returning False for any letters and symbols. However, Processing represents special keys (arrows and modifiers) by using numeric codes—in other words, integers, not strings. So, you use str(key) to convert any numeric codes to strings to prevent certain keypresses from crashing the app. If the key value is a digit, Python will subtract 1 from it and assign it to variable k. Because the swatches list is zero-indexed, color 1 is equal to swatches[0], and so forth. The final if statement verifies that the index value (k) is less than the length of the swatches list—in other words, a number between 0 and 5. The redraw() function updates the brush preview.

The paint app can paint in different colors, in strokes of varying thickness. Experiment with adding other features to your paint app.

Challenge #11: Adding Paint App Features

One of the most useful features you can add is a Clear button so you don't need to close and reopen the app when you want a blank, new canvas. You'll program a button that resets the canvas to dark blue.

Add a button labeled CLEAR to the palette:

```
. . .
def draw():
    . . .
    # clear button
    fill('#FFFFFF')
    text('CLEAR', 10, height-12)
```

This draws CLEAR, in the Ernest font, in the lower left corner of the display window (Figure 11-10).

Figure 11-10: The Clear button

You can use a `mouseClicked()` function to execute code when a mouse button is clicked, at the moment of release. Like the other mouse events, this code executes just once until you repeat the action. Add a `mouseClicked()` function to your code:

```
. . .
def mouseClicked():
    circle(width/2, height/2, width)
```

If you click anywhere in the display window, this code will draw a circle over the entire paint app. Now replace the `circle()` line with code that responds only to clicks over the Clear button and not anywhere outside that region. Additionally, this code must draw a dark blue square over the canvas area.

Once you have the Clear button working correctly, try adding a Save (As Image) button, an eraser, more swatches, or maybe even a color mixer. If you need help, you can find the solution at *https://github.com/tabreturn/processing.py-book/tree/master/chapter-11-mouse_and_keyboard_interaction/paint_app/*.

Summary

In this chapter, you learned how to add interactivity to your sketches by using mouse and keyboard input. You learned about Processing's system variables for those input devices, as well as their event functions that execute, just once, when a specific event occurs.

Processing supports a range of input devices, such as microphones, cameras, and game controllers, and I encourage you to explore those features. You can also connect an Arduino board to Processing if you want to build custom input devices.

In this chapter, you programmed a simple tool palette for selecting color swatches. Many software and web development projects require graphic interface development, and many graphical user interface (GUI) toolkits provide sets of ready-made widgets for things like buttons, checkboxes, sliders, drop-down lists, and text fields. Processing has GUI libraries to explore if you want to build more complex interfaces. For Python (outside of Processing), Tkinter, PyQt, and Kivy are a few options.

In the Afterword, I'll point you to other useful resources and suggest the next steps you might consider taking in your creative coding adventures.

AFTERWORD

Well done. You've reached the end of this book. You've come a long way since that first print() function! You've used Processing's Python Mode to delve into randomness, periodic motion, Euclidean vectors, interactivity, and more. You learned how to write code for generating patterns, animations, and data visualizations. You also learned to read in data from CSV and JSON files, and how to structure your programs by using object-orientated programming and modules. If you ever need to refer to the examples in this book, you can access them at *https://github.com/tabreturn/processing.py-book/* or look at *https://www.nostarch.com/Learn-Python-Visually/.* This repository includes solutions to the challenges in each chapter. But there's plenty more to explore.

Your newly acquired Processing, Python, and creative coding skills are your gateway to programming for an ever-expanding horizon of creative technologies, such as games, the web, augmented/virtual reality, and even visual effects for films. I'll suggest a few topics here that you might explore next—namely, more advanced Processing techniques, various Python frameworks, and different creative coding environments. Your next big coding project could range from highly expressive (like something you'd see in an art gallery) to highly functional, or anything in between.

You can head in many directions from here. To begin, take a look at other projects for inspiration—investigate how they're built, the techniques and programming languages involved, and so forth. Peruse the collection of works showcased on the Creative Applications Network website (*https://creativeapplications.net/*) and elsewhere on the web.

More Python for Processing

Processing has a large community of artists, geeks, tinkerers, designers, researchers, hobbyists, and educators who tend to gather on the official Processing forums at *https://discourse.processing.org/*. You can chat with community members, get help if you're stuck, and keep up-to-date with new developments and events. The site includes a dedicated category for Processing.py and a gallery section to share your creations.

The official Python Mode for Processing reference is available at *https://py.processing.org/reference/*. Each entry includes a description and brief code example. You can also find helpful tutorials in the Tutorials section of the website; the following are topics you may want to explore here:

- "Images and Pixels" so that you can manipulate graphics on a pixel level. You can even create your very own Photoshop-esque filters this way.
- Processing's 3D render mode, "P3D," for drawing in three dimensions using x-, y-, and z-coordinates, with texture and lighting effects.
- Processing's myriad libraries, for physics, GUIs, video, AI, audio, and more. You can find the link to the "Libraries" section on the main Processing page (*https://processing.org/*). The libraries are built for Processing Java Mode, but it's possible to get most of them working in Python Mode.

On that last point, you'll find many useful algorithms written in other languages, especially Processing's Java Mode. Seek assistance on the forums to help implement those in Python Mode, but after a while, you'll likely be able to translate Java code to Python by yourself. This also means you can look at Processing Java resources to learn new techniques for your Python Mode sketches.

The Nature of Code is a fantastic book by Daniel Shiffman for Processing's Java Mode, which you can read online at *https://natureofcode.com/book/*. It teaches you how to write code to simulate things that occur naturally in the physical world. Topics include vectors, forces, particles systems, physics, autonomous agents, fractals, evolutionary algorithms, and neural networks. You can find Python versions of the tasks at *https://github.com/nature-of-code/noc-examples-python/*.

More Python

Python is a general-purpose language, suitable for programming AI, games, simulations, web applications, and just about everything in between; for each domain, there are multiple Python libraries and frameworks to explore.

No Starch Press has published several great Python books. For game development, there's *Invent Your Own Computer Games with Python* by Al Sweigart. For some geeky fun, there's *Python Playground* by Mahesh Venkitachalam, and *Impractical Python Projects* by Lee Vaughan. *Python*

Crash Course, 2nd edition, by Eric Matthes is an in-depth Python book that also covers some game (Pygame), data visualization (matplotlib), and web (Django) development.

Recall that, at the time of this writing, Python Mode for Processing uses Python 2.7, but I've ensured that all the code in this book is Python 3–compatible. You might not even realize the difference when you switch to a Python 3 development environment.

Other Creative Coding Environments

If you'd like to learn a different programming language for creative coding, consider Processing's Java Mode, as well as Processing variants for JavaScript (*p5.js*) and Ruby (*JRubyArt*). Beyond Processing, there's openFrameworks for creative coding in the C++ language and *OPENRNDR* for Kotlin. This isn't everything, but that list should be enough to get you looking in the right places to discover more.

If you want to program devices that can interact with the real world, Arduino provides an open source platform for electronics projects. It's a relatively cheap, programmable development board that you can connect to sensors, motors, lights, and other electronic components. You can also get your Arduino board talking to your Processing sketches.

INDEX

translate() function *(continued)*
    Analog Clock challenge, 130
    cumulative nature of, 122
    drawing shapes by using list of color values, 140–141
    origin, 120
    transformations within draw() block, 122
triangle() function, 17
trigonometric functions, 190–192
trigonometry (trig), 188
Truchet, Sébastien, 97
Truchet tiles, 97–102
TrueType Font (TTF) files, 66
tuples, 166
typography, 62–67
    fonts, 63–64
    text functions, 64–67

U

upper() method, 60–61, 187
UpperCamelCase, 209
uppercase, converting characters to, 60–61
URLs
    domain, 59
    slice notation, 58–59
    subdomain, 59
    top-level domain, 59

V

values() method, 165
van Eyck, Jan, 179
variables, 19–21
    defined, 19
    global variables, 111–115
    naming, 20
    printing to console, 19–20
    system variables, 19

Vaughan, Lee, 256
vector graphics
    Bézier curves, 39–40
    vector graphics drawing software, 50–51
vectors, 224–233
    Euclidean vectors, 208, 224
    magnitude, 224
    PVector class, 225–233
        adding vectors, 228–229
        animation with, 226–233
        limiting vector magnitude, 231–233
        subtracting vectors, 229–231
    Pythagorean theorem, 225
    scalar value, 224
Venkitachalam, Mahesh, 256
vertex() function, 41
vertices, 40–50
    Bézier vertices, 43–50
        Chinese coin shapes, 46–50
        heart shapes, 45–46
        S-curves, 44–45
    defined, 40
    drawing squares with, 41–42

W

wavelength, 196
web APIs (web application programming interfaces), 169–170
while loops, 87–89
whitespace, 7
wraparound, 235

X

XML (Extensible Markup Language), 170

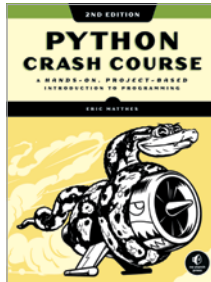*Learn Python Visually* is set in New Baskerville, Futura, Dogma, and TheSansMono Condensed.

RESOURCES

Visit *https://nostarch.com/Learn-Python-Visually/* for errata and more information.
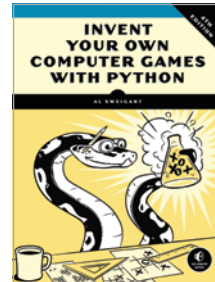
*More no-nonsense books from* **NO STARCH PRESS**

REAL WORLD PYTHON
A Hacker's Guide to Solving Problems with Code
*by* LEE VAUGHAN
360 PP., $34.95
ISBN 978-1-7185-0062-4

PYTHON CRASH COURSE, 2ND EDITION
A Hands-On, Project-Based Introduction to Programming
*by* ERIC MATTHES
544 PP., $39.95
ISBN 978-1-59327-928-8

INVENT YOUR OWN COMPUTER GAMES WITH PYTHON, 4TH EDITION
by AL SWEIGART
376 PP., $29.95
ISBN 978-1-59327-795-6
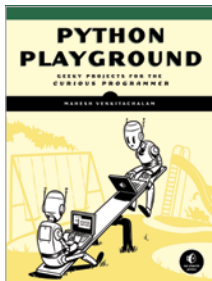
PYTHON PLAYGROUND
Geeky Projects for the Curious Programmer
by MAHESH VENKITACHALAM
352 PP., $29.95
ISBN 978-1-59327-604-1
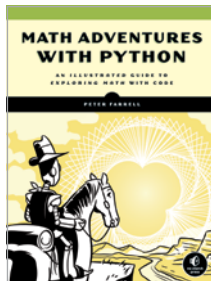
MATH ADVENTURES WITH PYTHON
An Illustrated Guide to Exploring Math With Code
by PETER FARRELL
304 PP., $29.95
ISBN 978-1-59327-867-0
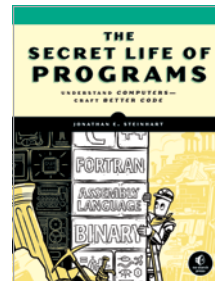
THE SECRET LIFE OF PROGRAMS
Understand Computers—Craft Better Code
by JONATHAN E. STEINHART
504 PP., $44.95
ISBN 978-1-59327-970-7

Never before has the world relied so heavily on the Internet to stay connected and informed. That makes the Electronic Frontier Foundation's mission—to ensure that technology supports freedom, justice, and innovation for all people—more urgent than ever.

For over 30 years, EFF has fought for tech users through activism, in the courts, and by developing software to overcome obstacles to your privacy, security, and free expression. This dedication empowers all of us through darkness. With your help we can navigate toward a brighter digital future.

ELECTRONIC FRONTIER FOUNDATION **EFF**
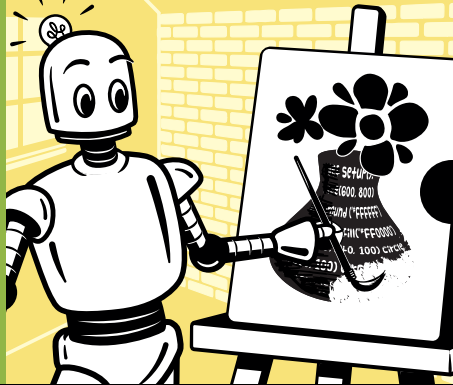
PYTHON FOR VISUAL LEARNERS

The perfect book for first-time programmers, *Learn Python Visually* introduces the fundamentals of computer coding within a visual, graphics-based context. Tristan Bunn's creative teaching approach will help you visualize core programming concepts as you make cool pictures, animations, and simulations using Python Mode for the open source Processing development environment.

From the very first chapter, you'll produce and manipulate colorful designs as Bunn walks you through a series of easy-to-follow graphical coding projects that grow increasingly complex. You'll progress from drawing with code to animating a bouncing DVD screensaver to creating interactive programs. Along the way, you'll encounter skill-building challenges on topics as diverse as video games, coffee, fine art, amoebas, and Pink Floyd.

You'll learn how to:

- Apply basic coding theories and concepts, like variables, data types, pixel coordinates, control flow, and algorithms
- Write code that produces drawings, patterns, animations, data visualizations, user interfaces, and simulations
- Use code to explore randomness, handle CSV and JSON data, and generate animations that employ trigonometry for periodic motion and Lissajous curves
- Define functions, reduce repetition, and make your code more modular
- Write classes and create objects to structure code more efficiently

If you've ever dreamed of coding for data visualizations, computer-generated art, games, or other creative technologies, *Learn Python Visually* is the entry point you need.

ABOUT THE AUTHOR

For over a decade, Tristan Bunn has worked on a diverse range of digital projects for varied clients. He's currently involved in lecturing, research, and work that blends code, interaction, interface design, and creativity.

FULL COLOR

THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

no starch press

$49.99 ($65.99 CDN)

ISBN 978-1-7185-0096-9