Invent Your Own Computer Games with Python
3rd Edition

By Al Sweigart

If you've downloaded this book from a torrent, it's probably out of date. Go to http://inventwithpython.com to download the latest version instead.

For Caro, with more love

than I ever knew I had.

A Note to Parents and Fellow Programmers

Thank you for reading this book. My motivation for writing it came from a gap I saw in today's literature for kids interested in learning to program. I started programming in the BASIC programming language with a book similar to this one.

During the course of writing this, I've realized how a modern language like Python has made programming far easier and versatile for a new generation of programmers. Python has a gentle learning curve while still being a serious language used by programmers professionally.

The current crop of programming books fall into two categories. First, books that didn't teach programming so much as "game creation software" or a dumbed-down languages to make programming "easy" to the point that it is no longer programming. Or second, they taught programming like a mathematics textbook: all principles and concepts with little application given to the reader. This book takes a different approach: show the source code for games right up front and explain programming principles from the examples.

I've also made this book available under the Creative Commons license, which allows you to make copies and distribute this book (or excerpts) with my full permission, as long as attribution to me is left intact and it is used for noncommercial purposes. (See the copyright page.) I want to make this book a gift to a world that has given me so much.

What's New in the 3ʳᵈ Edition?

The third edition features no new content since the second edition. However, the third edition has been streamlined to cover the same content with 20% fewer pages. Explanations have been expanded where needed and ambiguities clarified.

Chapter 9 was split into chapters 9 and 9½ to keep the chapter numbering the same.

The source code has intentionally been kept the same as the second edition to prevent confusion. If you've already read the second edition, there's no reason to read this book. However, if you are new to programming, or introducing a friend to programming, this third edition will make the process easier, smoother, and more fun.
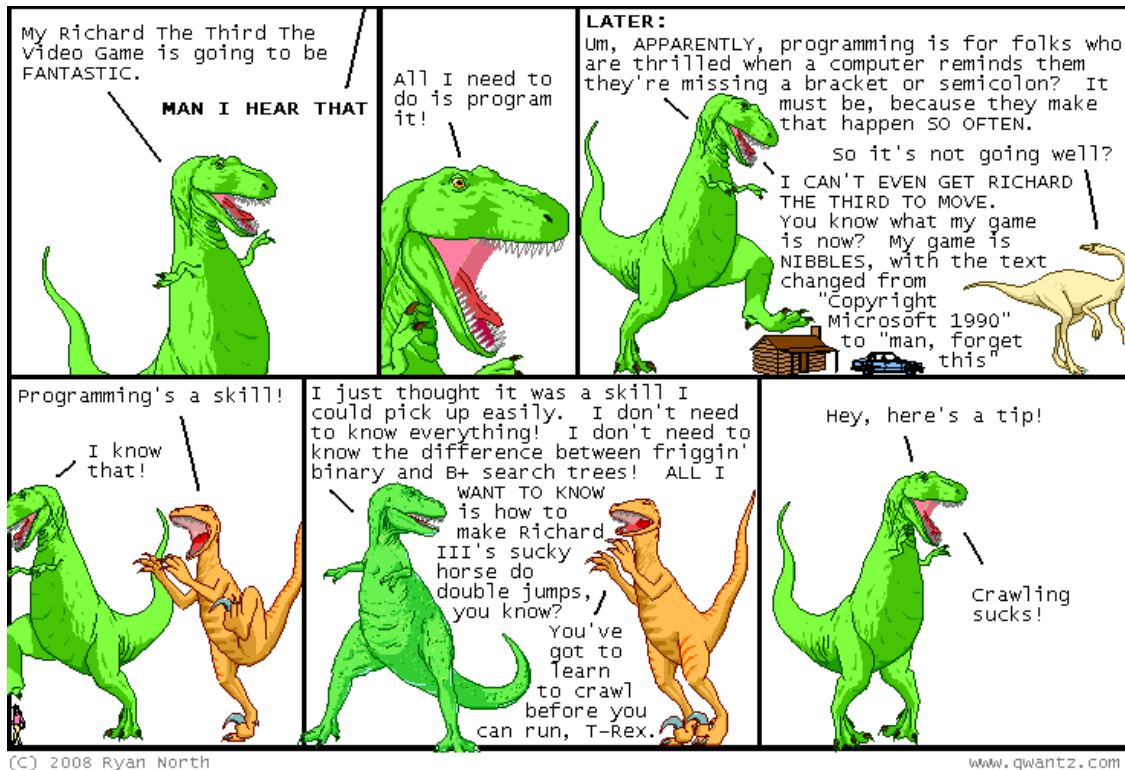
Who is this book for?

Programming isn't hard. But it is hard to find learning materials that teach you to do interesting things with programming. Other computer books go over many topics most newbie coders don't need. This book will teach you how to program your own computer games. You'll learn a useful skill and have fun games to show for it! This book is for:

- Complete beginners who want to teach themselves computer programming, even if they have no previous experience programming.
- Kids and teenagers who want to learn programming by creating games.
- Adults and teachers who wish to teach others programming.
- Anyone, young or old, who wants to learn how to program by learning a professional programming language.

TABLE OF CONTENTS

Chapter 1

INSTALLING PYTHON

> Topics Covered In This Chapter:
> * Downloading and installing the Python interpreter
> * How to use this book
> * The book's website at http://inventwithpython.com

Hello! This book teaches you how to program by making video games. Once you learn how the games in this book work, you'll be able to create your own games. All you'll need is a computer, some software called the Python interpreter, and this book. The Python interpreter is free to download from the Internet.

When I was a kid, a book like this one taught me how to write my first programs and games. It was fun and easy. Now as an adult, I still have fun programming and I get paid for it. But even if you don't become a computer programmer when you grow up, programming is a useful and fun skill to have.

Computers are incredible machines, and learning to program them isn't as hard as people think. If you can read this book, you can program a computer. A computer **program** is a bunch of instructions that the computer can understand, just like a storybook is a bunch of sentences understood by the reader. Since video games are nothing but computer programs, they are also made up of instructions.

To instruct a computer, you write a program in a language the computer understands. This book teaches a programming language named Python. There are many different programming languages including BASIC, Java, JavaScript, PHP, and C++.

When I was a kid, BASIC was a common first language to learn. However, new programming languages such as Python have been invented since then. Python is even easier to learn than BASIC! But it's still a serious programming language used by professional programmers. Many adults use Python in their work and when programming for fun.

The games you'll create from this book seem simple compared to the games for Xbox, PlayStation, or Nintendo. These games don't have fancy graphics because they're meant to teach coding basics. They're purposely simple so you can focus on learning to program. Games don't have to be complicated to be fun.

Downloading and Installing Python

You'll need to install software called the Python interpreter. The **interpreter** program understands the instructions you'll write in the Python language. I'll just refer to "the Python interpreter software" as "Python" from now on.

> **Important Note!** Be sure to install Python 3, and not Python 2. The programs in this book use Python 3, and you'll get errors if you try to run them with Python 2. It is so important I've added a cartoon penguin in Figure 1-1 to tell you to install Python 3 so you do not miss this message.

Figure 1-1: An incongruous penguin tells you to install Python 3.

On Windows, download the Python installer (the filename will end with *.msi*) and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. Select Install for All Users and then click **Next**.

2. Install to the *C:\Python34* folder by clicking **Next**.

3. Click **Next** to skip the Customize Python section.

On Mac OS X, download the *.dmg* file that's right for your version of OS X from the website and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. When the DMG package opens in a new window, double-click the *Python.mpkg* file. You may have to enter the administrator password.

2. Click **Continue** through the Welcome section and click **Agree** to accept the license.

3. Select HD Macintosh (or whatever name your hard drive has) and click Install.

If you're running Ubuntu, you can install Python from the Ubuntu Software Center by following these steps:

1. Open the Ubuntu Software Center.

2. Type *Python* in the search box in the top-right corner of the window.

3. Select **IDLE (using Python 3.4)**, or whatever is the latest version.

4. Click **Install**. You may have to enter the administrator password to complete the installation.

Starting IDLE

IDLE stands for **I**nteractive **D**eve**L**opment **E**nvironment. The development environment is like word processing software for writing Python programs. Starting IDLE is different on each operating system.

On Windows, click the Start button in the lower left corner, type "IDLE" and select **IDLE (Python GUI)**.

On Mac OS X, open the Finder window and click on **Applications**. Then click **Python 3.4**. Then click the IDLE icon.

On Ubuntu or Linux, open a terminal window and then type "idle3". You may also be able to click on **Applications** at the top of the screen. Then click **Programming** and **IDLE 3**.

The window that appears when you first run IDLE is the **interactive shell**, as shown in Figure 1-2. You can enter Python instructions into the interactive shell at the >>> prompt and Python will perform them. After displaying instruction results, a new >>> prompt will wait for your next instruction.

Figure 1-2: The IDLE program's interactive shell on Windows, OS X, and Ubuntu Linux.

How to Use this Book

Most chapters in this book will begin with a sample run of the chapter's featured program. This sample run shows you what the program looks like when you run it. The parts the user types in are shown as **bold** print.

Type the code for the program into IDLE's file editor yourself, rather than download or copy/paste it. You'll remember programming better if you take the time to type in the code.

Line Numbers and Spaces

When typing the source code from this book, do **not** type the line numbers at the start of each line. For example, if you see this in the book:

```
9. number = random.randint(1, 20)
```

You do not need to type the "9." on the left side, or the one space immediately following it. Just type it like this:

```
number = random.randint(1, 20)
```

Those numbers are there only so that this book can refer to specific lines in the program. They are not a part of the actual program's source code.

Aside from the line numbers, enter the code exactly as it appears. Notice that some of the lines of code are indented by four or eight spaces. Each character in IDLE is the same width, so you can count the number of spaces by counting the number of characters on the line above or below.

For example, the indented spaces here are marked with a ▪ black square so you can see them:

```
while guesses < 10:
▪▪▪▪if number == 42:
▪▪▪▪▪▪▪▪print('Hello')
```

Text Wrapping in This Book

Some instructions are too long to fit on one line on the page and will wrap around to the next line. When you type this code, enter it all on one line without pressing ENTER. You can tell when a new instruction starts by looking at the line numbers on the left. The example below has only two instructions:

```
1. print('This is the first instruction! xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxx')
```

```
2. print('This is the second instruction, not the third instruction.')
```

The first instruction wraps around and makes it look like three instructions in total. That's only because this book's pages aren't wide enough to fit the first instruction on one line.

Finding Help Online

This book's website is at http://inventwithpython.com. You can find several resources related to this book there. Several links in this book use the invpy.com domain name for shortened URLs.

The website at http://reddit.com/r/inventwithpython is a great place to ask programming questions related to this book. Post general Python questions to the LearnProgramming and LearnPython websites at http://reddit.com/r/learnprogramming and http://reddit.com/r/learnpython, respectively.

You can also email me your programming questions at *al@inventwithpython.com*.

Keep in mind there are smart ways to ask programming questions that help others help you. Be sure to read the Frequently Asked Questions sections these websites have about the proper way to post questions. When asking programming questions, do the following:

- If you are typing out the programs in this book but getting an error, first check for typos with the online diff tool at http://invpy.com/diff. Copy and paste your code into the diff tool to find any differences from the book's code in your program.
- Explain what you are trying to do when you explain the error. This will let your helper know if you are on the wrong path entirely.
- Copy and paste the entire error message and your code.
- Search the Web to see whether someone else has already asked (and answered) your question.
- Explain what you've already tried to do to solve your problem. This tells people you've already put in some work to try to figure things out on your own.
- Be polite. Don't demand help or pressure your helpers to respond quickly.

Asking someone, "Why isn't my program working?" doesn't tell them anything. Tell them what you are trying to do, the exact error you are getting, and your operating system and version.

Chapter 2

THE INTERACTIVE SHELL

Topics Covered In This Chapter:
- Integers and Floating Point Numbers
- Expressions
- Values
- Operators
- Evaluating Expressions
- Storing Values in Variables

Before you can make games, you need to learn a few basic programming concepts. You won't make games in this chapter, but learning these concepts is the first step to programming video games. You'll start by learning how to use Python's interactive shell.

Some Simple Math Stuff

Open IDLE using the steps in Chapter 1, then get Python to solve some simple math stuff. The interactive shell can work just like a calculator. Type 2 + 2 into the interactive shell at the >>> prompt and press the ENTER key on your keyboard. (On some keyboards, this is the RETURN key.) Figure 2-1 shows how the interactive shell responds with the number 4.
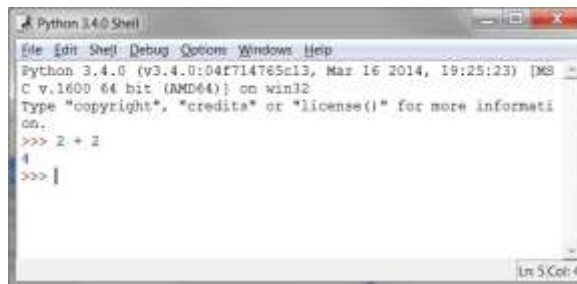
Figure 2-1: Enter 2+2 into the interactive shell.

This math problem is a simple programming instruction. The + sign tells the computer to add the numbers 2 and 2. Table 2-1 lists the other math symbols available in Python. The - sign will subtract numbers. The * asterisk will multiply numbers. The / slash will divide numbers.

Table 2-1: The various math operators in Python.

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division

When used in this way, +, -, *, and / are called **operators**. Operators tell Python what to do with the numbers surrounding them.

Integers and Floating Point Numbers

Integers (or **ints** for short) are whole numbers such as 4, 99, and 0. **Floating point numbers** (or **floats** for short) are fractions or numbers with decimal points like 3.5, 42.1 and 5.0. In Python, the number 5 is an integer, but 5.0 is a float. These numbers are called **values**.

Expressions

These math problems are examples of expressions. Computers can solve millions of these problems in seconds. **Expressions** are made up of values (the numbers) connected by operators (the math signs). Try entering some of these math problems into the interactive shell, pressing the ENTER key after each one.

```
2+2+2+2+2
8*6
10-5+6
2    +         2
```

After you type in the above instructions, the interactive shell will look like Figure 2-2.
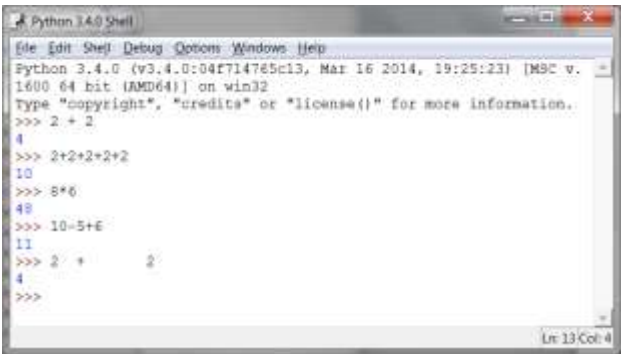
Figure 2-2: What the IDLE window looks like after entering instructions.
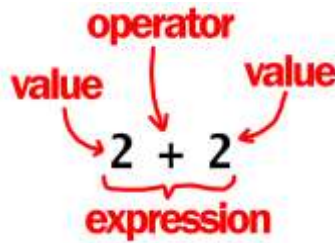
Figure 2-3: An expression is a made up of values and operators.

In the 2 + 2 example, notice that there can be any amount of spaces between the values and operators. However, always start instructions at the beginning of the line when entering them into the interactive shell.

Evaluating Expressions

When a computer solves the expression 10 + 5 and gets the value 15, it has **evaluated** the expression. Evaluating an expression *reduces the it to a single value*, just like solving a math problem reduces the problem to a single number: the answer. The expressions 10 + 5 and 10 + 3 + 2 both evaluate to 15.

Expressions can be of any size, but they will always evaluate down to a single value. Even single values are expressions: The expression 15 evaluates to the value 15. For example, the expression 8 * 3 / 2 + 2 + 7 – 9 will evaluate down to the value 12.0 through the following steps:

```
8 * 3 / 2 + 2 + 7 – 9
                 ▼
  24 / 2 + 2 + 7 – 9
                 ▼
    12.0 + 2 + 7 – 9
                 ▼
        14.0 + 7 – 9
                 ▼
            21.0 – 9
                 ▼
                12.0
```

You don't see all of these steps in the interactive shell. The interactive shell does them and just shows you the results:

```
>>> 8 * 3 / 2 + 2 + 7 – 9
12.0
```

Notice that the / division operator evaluates to a float value, as in 24 / 2 evaluating to 12.0. Math operations with float values also evaluate to float values, as in 12.0 + 2 evaluating to 14.0.

Syntax Errors

If you enter 5 + into the interactive shell, you'll get an error message.

```
>>> 5 +
SyntaxError: invalid syntax
```

This error happened because 5 + isn't an expression. Expressions have values connected by operators. But the + operator expects a value after the + sign. An error message appears when this value is missing.

SyntaxError means Python doesn't understand the instruction because you typed it incorrectly. A lot of computer programming isn't just telling the computer what to do, but also knowing how to tell it.

Don't worry about making mistakes though. Errors don't damage your computer. Just retype the instruction correctly into the interactive shell at the next >>> prompt.

Storing Values in Variables

You can save the value an expression evaluates to so you can use it later by storing them in **variables**. Think of variables like a box that can hold a value.

An **assignment statement** instruction will store a value inside a variable. Type the name for the variable, followed by the = sign (called the **assignment operator**), and then the value to store in the variable. For example, enter spam = 15 into the interactive shell:

```
>>> spam = 15
>>>
```

The spam variable's box will have the value 15 stored in it, as shown in Figure 2-4. The name "spam" is the label on the box (so Python can tell variables apart) and the value is written on a small note inside the box.

When you press ENTER you won't see anything in response. In Python, the instruction executed was successful if no error message appears. The >>> prompt will appear so you can type in the next instruction.

Figure 2-4: Variables are like boxes that can hold values in them.

Unlike expressions, **statements** are instructions that do not evaluate to any value. This is why there's no value displayed on the next line in the interactive shell after spam = 15. If you are confused about which instructions are expressions and which are statements, remember that expressions evaluate to a single value. Any other kind of instruction is a statement.

Variables store values, not expressions. For example, consider the expression in the statements spam = 10 + 5 and spam = 10 + 7 – 2. They both evaluate to 15. The end result is the same: Both assignment statements store the value 15 in the variable spam.

The first time a variable is used in an assignment statement, Python will create that variable. To check what value is in a variable, type the variable name into the interactive shell:

```
>>> spam = 15
>>> spam
15
```

The expression spam evaluates to the value inside the spam variable: 15. You can use variables in expressions. Try entering the following in the interactive shell:

```
>>> spam = 15
>>> spam + 5
20
```

You've set the value of the variable spam to 15, so writing spam + 5 is like writing the expression 15 + 5. Here are the steps of spam + 5 being evaluated:

```
spam + 5
     ▼
 15  + 5
     ▼
    20
```

You cannot use a variable before an assignment statement creates it. Python will give you a NameError because no such variable by that name exists yet. Mistyping the variable name also causes this error:

```
>>> spam = 15
>>> spma
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    spma
NameError: name 'spma' is not defined
```

The error appeared because there's spam variable but no variable named spma.

You can change the value stored in a variable by entering another assignment statement. For example, try entering the following into the interactive shell:

```
>>> spam = 15
>>> spam + 5
20
>>> spam = 3
>>> spam + 5
8
```

When you first enter spam + 5, the expression evaluates to 20 because you stored 15 inside spam. However, when you enter spam = 3, the value 15 is replaced, or **overwritten**, with the value 3. Now when you enter spam + 5, the expression evaluates to 8 because the value of spam is now 3. Overwriting is shown in Figure 2-5.

Figure 2-5: The 15 value in spam being overwritten by the 3 value.

You can even use the value in the spam variable to assign a new value to spam:

```
>>> spam = 15
>>> spam = spam + 5
20
```

The assignment statement spam = spam + 5 is like saying, "the new value of the spam variable will be the current value of spam plus five." Keep increasing the value in spam by 5 several times by entering the following into the interactive shell:

```
>>> spam = 15
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam
30
```

Using More Than One Variable

Create as many variables as you need in your programs. For example, let's assign different values to two variables named eggs and bacon, like so:

```
>>> bacon = 10
>>> eggs = 15
```

Now the bacon variable has 10 inside it, and eggs has 15 inside it. Each variable is its own box with its own value, like in Figure 2-6.
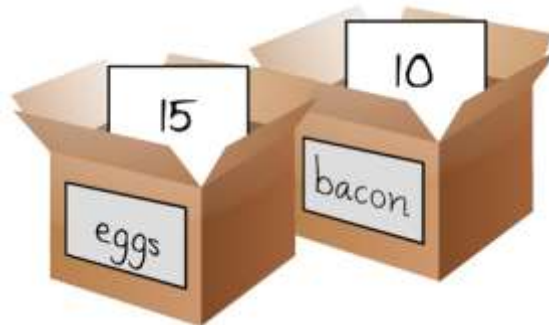
Figure 2-6: The "bacon" and "eggs" variables have values stored in them.

Try entering spam = bacon + eggs into the interactive shell, then check the new value of spam:

```
>>> bacon = 10
>>> eggs = 15
>>> spam = bacon + eggs
>>> spam
```

```
25
```

The value in `spam` is now 25. When you added `bacon` and `eggs` you are adding their values, which are 10 and 15, respectively. Variables contain values, not expressions. The `spam` variable was assigned value 25, and not the expression `bacon + eggs`. After the `spam = bacon + eggs` assignment statement, changing `bacon` or `eggs` does not affect `spam`.

Summary

In this chapter, you learned the basics about writing Python instructions. Python needs you to tell it exactly what to do in a strict way. Computers don't have common sense and only understand specific instructions.

Expressions are values (such as 2 or 5) combined with operators (such as + or -). Python can evaluate expressions, that is, reduce the expression to a single value. You can store values inside of variables so that your program can remember them and use them later.

There are many other types of operators and values in Python. In the next chapter, you'll go over some more basic concepts and write your first program. You'll learn about working with text in expressions. Python isn't limited to just numbers; it's more than a calculator!

Chapter 3
WRITING PROGRAMS

Topics Covered In This Chapter:
- Flow of execution
- Strings
- String concatenation
- Data types (such as strings or integers)
- Using the file editor to write progams
- Saving and running programs in IDLE
- The `print()` function
- The `input()` function
- Comments
- Case-sensitivity

That's enough math for now. Now let's see what Python can do with text. In this chapter, you'll learn how to store text in variables, combine text, and display text on the screen.

Almost all programs display text to the user, and the user enters text into your programs through the keyboard. You'll also make your first program in this chapter. This program displays the greeting, "Hello World!" and asks for the user's name.

Strings

In Python, text values are called **strings**. String values can be used just like integer or float values. You can store strings in variables. In code, string values start and end with a single quote ('). Try entering this code into the interactive shell:

```
>>> spam = 'hello'
```

The single quotes tell Python where the string begins and ends. They are not part of the string value's text. Now if you type spam into the interactive shell, you will see the contents of the spam variable. Remember, Python evaluates variables to the value stored inside the variable. In this case, this is the string 'hello':

```
>>> spam = 'hello'
>>> spam
'hello'
```

Strings can have any keyboard character in them and can be as long as you want. These are all examples of strings:

```
'hello'
'Hi there!'
'KITTENS'
'7 apples, 14 oranges, 3 lemons'
'Anything not pertaining to elephants is irrelephant.'
'A long time ago, in a galaxy far, far away...'
'O*&#wY%*&OCfsdYO*&gfC%YO*&%3yc8r2'
```

String Concatenation

String values can combine with operators to make expressions, just like integer and float values do. You can combine two strings with the + operator. This is **string concatenation**. Try entering 'Hello' + 'World!' into the interactive shell:

```
>>> 'Hello' + 'World!'
'HelloWorld!'
```

The expression evaluates to a single string value, `'HelloWorld!'`. There is no space between the words because there was no space in either of the two concatenated strings, unlike this example:

```
>>> 'Hello ' + 'World!'
'Hello World!'
```

The + operator works differently on string and integer values because they are different data types. All values have a data type. The data type of the value `'Hello'` is a string. The data type of the value 5 is an integer. The data type tells Python what operators should do when evaluating expressions. The + operator will concatenate string values but add integer and float values.

Writing Programs in IDLE's File Editor

Until now, you've been typing instructions into IDLE's interactive shell one at a time. When you write programs though, you type in several instructions and have them run all at once. Let's write your first program!

IDLE has another part called the **file editor**. Click on the **File** menu at the top of the interactive shell window. Then select **New Window**. A blank window will appear for you to type your program's code into, like in Figure 3-1.
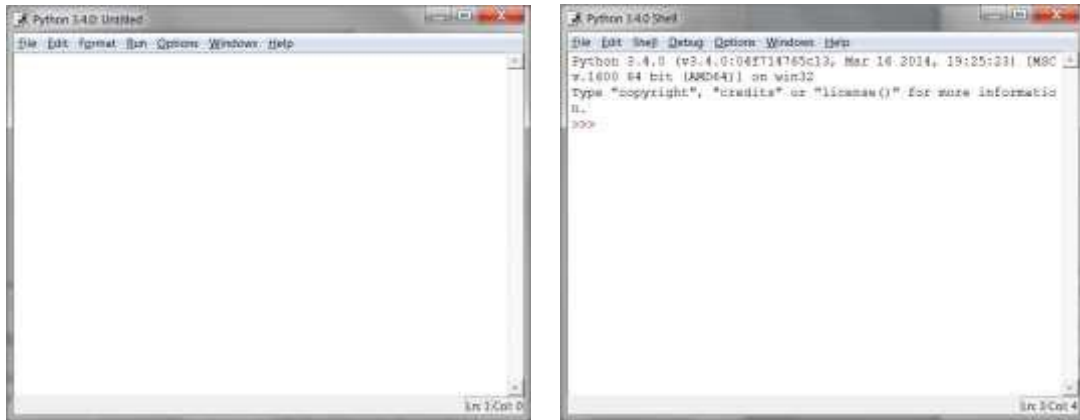
Figure 3-1: The file editor window (left) and the interactive shell window (right).

The two windows look similar, but just remember this: The **interactive shell window** will have the >>> prompt. The **file editor window** will not.

Hello World!

It's traditional for programmers to make their first program display "Hello world!" on the screen. You'll create your own Hello World program now.

When you enter your program, don't enter the numbers at the left side of the code. They're there so this book can refer to code by line number. The bottom-right corner of the file editor window will tell you where the blinking cursor is. Figure 3-2 shows that the cursor is on line 1 and column 0.
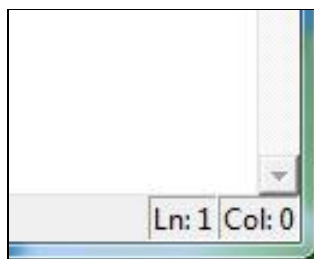
Figure 3-2: The bottom right of the file editor window tells you what line the cursor is on.

hello.py

Enter the following text into the new file editor window. This is the program's **source code**. It contains the instructions Python will follow when the program is run.

> **IMPORTANT NOTE!** The programs in this book will only run on Python 3, not Python 2. When the IDLE window starts, it will say something like "Python 3.4.2" at the top. If you have Python 2 installed, you can have Python 3 installed at the same time. To download Python 3, go to https://python.org/download/.

```
                                                                    hello.py
1. # This program says hello and asks for my name.
2. print('Hello world!')
3. print('What is your name?')
4. myName = input()
5. print('It is good to meet you, ' + myName)
```

The IDLE program will write different types of instructions with different colors. After you're done typing the code, the window should look like this:
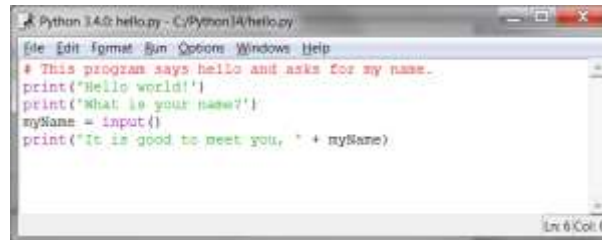
Figure 3-3: The file editor window will look like this after you type in the code.

Saving Your Program

Once you've entered your source code, save it by clicking on **File ► Save As**. Or press Ctrl-S to save with a keyboard shortcut. Figure 3-4 shows the Save As window that will open. Enter *hello.py* in the **File name** text field then click **Save**.
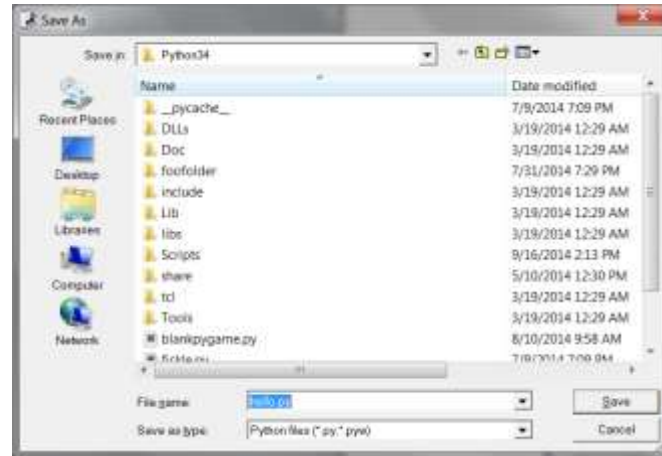
Figure 3-4: Saving the program.

You should save your programs often while you type them. That way, if the computer crashes or you accidentally exit from IDLE you won't lose much work.

Opening The Programs You've Saved

To load your previously saved program, click **File ► Open**. Choose the file in the window that appears and click the **Open** button. Your saved *hello.py* program will open in the File Editor window.

Now it's time to run the program. Click **File ► Run ► Run Module** or just press F5 from the file editor window. Your program will run in the interactive shell window.

Enter your name when the program asks for it. This will look like Figure 3-5.
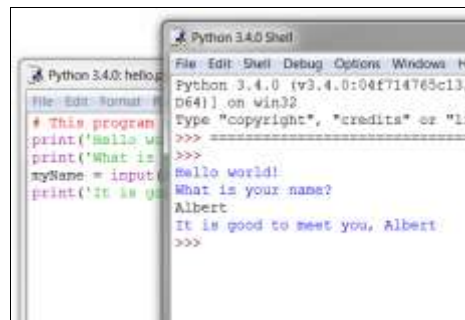
Figure 3-5: The interactive shell after running hello.py.

When you type your name and push ENTER, the program will greet you by name. Congratulations! You've written your first program and are now a computer programmer. Press F5 again to run the program a second time and enter another name.

If you got an error, compare your code to this book's code with the online diff tool at http://invpy.com/diff. Copy and paste your code from the file editor into the web page and click the **Compare** button. This tool will highlight any differences between your code and the code in this book, like in Figure 3-6.
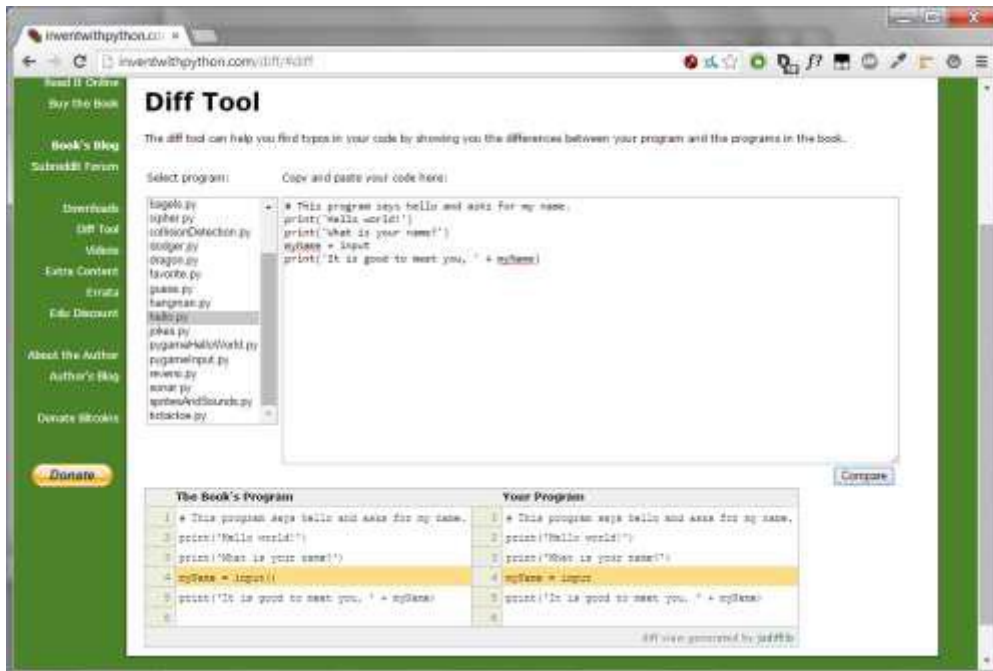
Figure 3-6: The diff tool at http://invpy.com/diff

While coding, if you get a NameError that looks like this:

```
Hello world!
What is your name?
Albert
Traceback (most recent call last):
  File "C:/Python26/test1.py", line 4, in <module>
    myName = input()
  File "<string>", line 1, in <module>
NameError: name 'Albert' is not defined
```

...that means you are using Python 2, instead of Python 3. Install a version of Python 3 from https://python.org/download. Re-run the program with Python 3.

How the "Hello World" Program Works

Each line of code is an instruction interpreted by Python. These instructions make up the program. A computer program's instructions are like the steps in a cookbook recipe. Each instruction executes in order, beginning from the top of the program and going down the list of instructions.

The step Python is at in the program is called the **execution**. When the program starts, the execution is at the first instruction. After executing the instruction, the execution moves down to the next instruction.

Let's look at each line of code to see what it's doing. We'll begin with line number 1.

Comments

```
1. # This program says hello and asks for my name.
```

This instruction is a **comment**. Any text following a **#** sign (called the **pound sign**) is a comment. Comments are not for Python, but for you, the programmer. Python ignores comments. Comments are the programmer's notes about what the code does. You can write anything in a comment. To make it easier to read the source code, this book prints comments in a light gray-colored text.

Programmers usually put a comment at the top of their code to give their program a title.

Functions

A **function** is kind of like a mini-program inside your program. Functions contain several instructions to execute when the function is called. Python provides some built-in functions already. Two functions, `print()` and `input()`, are described next. The great thing about functions is that you only need to know what the function does, but not how it does it.

A **function call** is an instruction that tells Python to run the code inside a function. For example, your program calls the `print()` function to display a string on the screen. The `print()` function takes the string you type between the parentheses as input and displays the text on the screen.

To display `Hello world!` on the screen, type the `print` function name, followed by an opening parenthesis, followed by the `'Hello world!'` string and a closing parenthesis.

The `print()` function

```
2. print('Hello world!')
```

```
3. print('What is your name?')
```

Lines 2 and 3 are calls to the print() function. A value between the parentheses in a function call is an **argument**. The argument on line 2's print() function call is 'Hello world!'. The argument on line 3's print() function call is 'What is your name?'. This is called **passing** the argument to the print() function.

In this book, function names have parentheses at the end. This makes it clear that print() means this book is talking about a function named print(), and not a variable named print. This is like the quotes around the number '42' telling Python that you are talking about a string '42' and not an integer 42.

The *input()* function

```
4. myName = input()
```

Line 4 is an assignment statement with a variable (myName) and a function call (input()). When input() is called, the program waits for the user to enter text. The text string that the user enters becomes the value that the function call evaluates to. Function calls can be used in expressions anywhere a value can be used.

The value that the function call evaluates to is called the **return value**. (In fact, "the value a function call returns" means the same thing as "the value a function call evaluates to".) In this case, the return value of the input() function is the string that the user typed in-their name. If the user typed in "Albert", the input() function call evaluates to the string 'Albert'. The evaluation looks like this:

```
myName = input()
          ▼
myName = 'Albert'
```

This is how the string value 'Albert' gets stored in the myName variable.

Using Expressions in Function Calls

```
5. print('It is good to meet you, ' + myName)
```

The last line is another print() function call. The expression 'It is good to meet you, ' + myName in between the parentheses of print(). However, arguments are always single values. Python will first evaluate this expression and then pass that value as the argument. If 'Albert' is stored in myName, the evaluation looks like this:

```
print('It is good to meet you, ' + myName)
                        ▼
print('It is good to meet you, ' + 'Albert')
                        ▼
print('It is good to meet you, Albert')
```

This is how the program greets the user by name.

Ending the Program

Once the program executes the last line, it **terminates** or **exits**. This means the program stops running. Python forgets all of the values stored in variables, including the string stored in myName. If you run the program again and enter a different name, the program will think that is your name.

```
Hello world!
What is your name?
Carolyn
It is good to meet you, Carolyn
```

Remember, the computer does exactly what you program it to do. Computers are dumb and just follow the instructions you give it exactly. The computer doesn't care if you type in your name, someone else's name, or just something silly. Type in anything you want. The computer will treat it the same way:

```
Hello world!
What is your name?
poop
It is good to meet you, poop
```

Variable Names

Giving variables descriptive names makes it easier to understand what a program does. Imagine if you were moving to a new house and you labeled every moving box "Stuff". That wouldn't be helpful at all!

Instead of myName, you could have called this variable abrahamLincoln or nAmE. Python doesn't care. It will run the program just the same.

Variable names are case-sensitive. **Case-sensitive** means the same variable name in a different case is considered a different variable. So spam, SPAM, Spam, and sPAM are four different variables in Python. They each contain their own separate values. It's a bad idea to have differently cased variables in your program. Use descriptive names for your variables instead.

Variable names are usually lowercase. If there's more than one word in the variable name, capitalize each word after the first. This makes your code more readable. For example, the variable name `whatIHadForBreakfastThisMorning` is much easier to read than `whatihadforbreakfastthismorning`. This is a **convention**: an optional but standard way of doing things in Python programming.

Short variable names are better than long names: `breakfast` or `foodThisMorning` is more readable than `whatIHadForBreakfastThisMorning`.

This book's interactive shell examples use variable names like `spam`, `eggs`, `ham`, and `bacon`. This is because the variable names in these examples don't matter. However, this book's programs all use descriptive names. Your programs should use descriptive variable names too.

Summary

Once you learn about strings and functions, you can start making programs that interact with users. This is important because text is the main way the user and the computer will communicate with each other. The user enters text through the keyboard with the `input()` function. The computer will display text on the screen with the `print()` function.

Strings are just values of a new data type. All values have a data type, and there are many data types in Python. The + operator can concatenate strings.

Functions are used to carry out some complicated instruction as part of your program. Python has many built-in functions that you'll learn about in this book. Function calls can be used in expressions anywhere a value is used.

The instruction in your program that Python is currently at is called the execution. In the next chapter, you'll learn more about making the execution move in ways other than just straight down the program. Once you learn this, you'll be ready to create games.

Chapter 4

GUESS THE NUMBER

Topics Covered In This Chapter:
- `import` statements
- Modules
- `while` statements
- Conditions
- Blocks
- Booleans
- Comparison operators
- The difference between = and ==
- `if` statements
- The `break` keyword
- The `str()` and `int()` and `float()` functions
- The `random.randint()` function

In this chapter, you're going to make a "Guess the Number" game. The computer will think of a random number from 1 to 20, and ask you to guess it. The computer will tell you if each guess is too high or too low. You win if you can guess the number within six tries.

This is a good game to code because it uses random numbers, loops, and input from the user in a short program. You'll learn how to convert values to different data types, and why you would need to do this. Since this program is a game, we'll call the user the **player**. But "user" would be correct too.

Sample Run of Guess the Number

Here's what the program looks like to the player when run. The text that the player types in is in **bold**.

```
Hello! What is your name?
Albert
Well, Albert, I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too high.
Take a guess.
2
```

```
Your guess is too low.
Take a guess.
4
Good job, Albert! You guessed my number in 3 guesses!
```

Source Code of Guess the Number

Open a new file editor window by clicking on the **File ► New Window**. In the blank window that appears, type in the source code and save it as *guess.py*. Then run the program by pressing **F5**. When you enter this code into the file editor, be sure to pay attention to the spacing at the front of some of the lines. Some lines have four or eight spaces of indentation.

> **IMPORTANT NOTE!** The programs in this book will only run on Python 3, not Python 2. When the IDLE window starts, it will say something like "Python 3.4.2" at the top. If you have Python 2 installed, you can have Python 3 installed at the same time. To download Python 3, go to https://python.org/download/.

If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/guess.

```
                                                             guess.py
 1. # This is a guess the number game.
 2. import random
 3.
 4. guessesTaken = 0
 5.
 6. print('Hello! What is your name?')
 7. myName = input()
 8.
 9. number = random.randint(1, 20)
10. print('Well, ' + myName + ', I am thinking of a number between 1 and 20.')
11.
12. while guessesTaken < 6:
13.     print('Take a guess.') # There are four spaces in front of print.
14.     guess = input()
15.     guess = int(guess)
16.
17.     guessesTaken = guessesTaken + 1
18.
19.     if guess < number:
20.         print('Your guess is too low.') # There are eight spaces in front
of print.
21.
22.     if guess > number:
```

```
23.          print('Your guess is too high.')
24.
25.      if guess == number:
26.          break
27.
28. if guess == number:
29.      guessesTaken = str(guessesTaken)
30.      print('Good job, ' + myName + '! You guessed my number in ' +
guessesTaken + ' guesses!')
31.
32. if guess != number:
33.      number = str(number)
34.      print('Nope. The number I was thinking of was ' + number)
```

`import` statements

```
1. # This is a guess the number game.
2. import random
```

The first line is a comment. Remember that Python will ignore everything after the # sign. This just reminds us what this program does.

The second line is an **import statement**. Remember, statements are instructions that perform some action but don't evaluate to a value like expressions do. You've already seen statements: assignment statements store a value in a variable.

While Python includes many built-in functions, some functions exist in separate programs called **modules**. You can use these functions by importing their modules into your program with an `import` statement.

Line 2 imports the module named `random` so that the program can call `random.randint()`. This function will come up with a random number for the user to guess.

```
4. guessesTaken = 0
```

Line 4 creates a new variable named `guessesTaken`. You'll store the number of guesses the player has made in this variable. Since the player hasn't made any guesses at this point in the program, store the integer 0 here.

```
6. print('Hello! What is your name?')
7. myName = input()
```

Lines 6 and 7 are the same as the lines in the Hello World program that you saw in Chapter 3. Programmers often reuse code from their other programs to save themselves work.

Line 6 is a function call to the `print()` function. Remember that a function is like a mini-program inside your program. When your program calls a function, it runs this mini-program. The code inside the `print()` function displays the string argument you passed it on the screen.

Line 7 lets the user type in their name and stores it in the `myName` variable. (Remember, the string might not really be the player's name. It's just whatever string the player typed. Computers are dumb and just follow their instructions no matter what.)

The `random.randint()` Function

```
9. number = random.randint(1, 20)
```

Line 9 calls a new function named `randint()` and stores the return value in `number`. Remember, function calls can be part of expressions because they evaluate to a value.

The `randint()` function is provided by the `random` module, so you must precede it with `random.` (don't forget the period!) to tell Python that the function `randint()` is in the `random` module.

The `randint()` function will return a random integer between (and including) the two integer arguments you pass to it. Line 9 passes `1` and `20` between the parentheses separated by commas that follow the function name. The random integer that `randint()` returns is stored in a variable named `number`; this is the secret number the player is trying to guess.

Just for a moment, go back to the interactive shell and enter `import random` to import the `random` module. Then enter `random.randint(1, 20)` to see what the function call evaluates to. It will return an integer between `1` and `20`. Repeat the code again and the function call will return a different integer. The `randint()` function returns random integer each time, just as rolling dice you'll get a random number each time:

```
>>> import random
>>> random.randint(1, 20)
12
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
7
```

Use the `randint()` function when you want to add randomness to your games. You'll use randomness in many games. (Think of how many board games use dice.)

You can also try different ranges of numbers by changing the arguments. For example, enter `random.randint(1, 4)` to only get integers between 1 and 4 (including both 1 and 4). Or try `random.randint(1000, 2000)` to get integers between 1000 and 2000.

For example, enter the following into the interactive shell. The results you get when you call the `random.randint()` function will probably be different (it is random, after all).

```
>>> random.randint(1, 4)
3
>>> random.randint(1000, 2000)
1294
```

You can change the game's code slightly to make the game behave differently. Try changing line 9 and 10 from this:

```
 9. number = random.randint(1, 20)
10. print('Well, ' + name + ', I am thinking of a number between 1 and 20.')
```

…into these lines:

```
 9. number = random.randint(1, 100)
10. print('Well, ' + name + ', I am thinking of a number between 1 and 100.')
```

And now the computer will think of an integer between 1 and 100 instead of 1 and 20. Changing line 9 will change the range of the random number, but remember to change line 10 so that the game also tells the player the new range instead of the old one.

Welcoming the Player

```
10. print('Well, ' + myName + ', I am thinking of a number between 1 and 20.')
```

On line 10 the `print()` function welcomes the player by name, and tells them that the computer is thinking of a random number.

It may look like there's more than one string argument in line 10, but look at the line carefully. The plus signs concatenate the three strings to evaluate down to one string. And that one string is the argument passed to the `print()` function. If you look closely, you'll see that the commas are inside the quotes and part of the strings themselves.

Loops

```
12. while guessesTaken < 6:
```

Line 12 is a `while` statement, which indicates the beginning of a `while` loop. **Loops** let you execute code over and over again. However, you need to learn a few other concepts first before learning about loops. Those concepts are blocks, Booleans, comparison operators, conditions, and the `while` statement.

Blocks

Several lines of code can be grouped together in a block. Every line in a **block** of code has the same minimum amount of indentation. You can tell where a block begins and ends by looking at the number of spaces at the front of the lines. This is the line's **indentation**.

A block begins when a line's indentation increases (usually by four spaces). Any following line also indented by four spaces is part of the block. The block ends when there's a line of code with the same indentation before the block started. This means blocks can exist within other blocks. Figure 4-1 is a diagram of code with the blocks outlined and numbered.

In Figure 4-1, line 12 has no indentation and isn't inside any block. Line 13 has an indentation of four spaces. Since this indentation is larger than the previous line's indentation, a new block has started. This block is labeled (1) in Figure 4-1. This block will continue until a line with zero spaces (the original indentation before the block began). Blank lines are ignored.

Line 20 has an indentation of eight spaces. Eight spaces is more than four spaces, which starts a new block. This block is labeled (2) in Figure 4-1. This block is inside of another block.

Figure 4-1: Blocks and their indentation. The black dots represent spaces.

Line 22 has only four spaces. Because the indentation has decreased, you know that block has ended. Line 20 is the only line in that block. Line 22 is in the same block as the other lines with four spaces.

Line 23 increases the indentation to eight spaces, so again a new block has started. It is labeled (3) in Figure 4-1.

To recap, line 12 isn't in any block. Lines 13 to 23 all in one block marked (1). Line 20 is in a block in a block marked as (2). Line 23 is the only line in another block in a block marked as (3).

The Boolean Data Type

The Boolean data type has only two values: `True` or `False`. These values must be typed with a capital "T" and "F". The rest of the value's name must be in lowercase. You will use Boolean values (called **bools** for short) with comparison operators to form conditions. (Conditions are explained later.)

For example, try storing the Boolean values in variables:

```
>>> spam = True
>>> eggs = False
```

The data types that have been introduced so far are integers, floats, strings, and now bools. Every value in Python belongs to one data type.

Comparison Operators

Line 12 has a `while` statement:

```
12. while guessesTaken < 6:
```

The expression that follows the `while` keyword (the `guessesTaken < 6` part) contains two values (the value in the variable `guessesTaken`, and the integer value 6) connected by an operator (the `<` "less than" sign). The `<` sign is a **comparison operator**.

Comparison operators compare two values and evaluate to a `True` or `False` Boolean value. A list of all the comparison operators is in Table 4-1.

Table 4-1: Comparison operators.

Operator Sign	Operator Name
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

You've already read about the +, -, *, and / math operators. Like any operator, the comparison operators combine with values to form expressions such as `guessesTaken < 6`.

Conditions

A **condition** is an expression that combines two values with a comparison operator (such as `<` or `>`) and evaluates to a Boolean value. A condition is just another name for an expression that evaluates to `True` or `False`. Conditions are used in `while` statements (and a few other instructions, explained later.)

For example, the condition `guessesTaken < 6` asks, "is the value stored in `guessesTaken` less than the number 6?" If so, then the condition evaluates to `True`. If not, the condition evaluates to `False`.

In the case of the "Guess the Number" program, on line 4 you stored the value 0 in `guessesTaken`. Because 0 is less than 6, this condition evaluates to the Boolean value of `True`. The evaluation would look like this:

```
guessesTaken < 6
        ▼
     0 < 6
        ▼
       True
```

Experiment with Booleans, Comparison Operators, and Conditions

Enter the following expressions in the interactive shell to see their Boolean results:

```
>>> 0 < 6
True
>>> 6 < 0
False
>>> 50 < 10
```

```
False
>>> 10 < 11
True
>>> 10 < 10
False
```

The condition `0 < 6` returns the Boolean value `True` because the number `0` is less than the number `6`. But because `6` isn't less than `0`, the condition `6 < 0` evaluates to `False`. `50` isn't less than `10`, so `50 < 10` is `False`. `10` is less than `11`, so `10 < 11` is `True`.

Notice that `10 < 10` evaluates to `False` because the number `10` isn't smaller than the number `10`. They are the same size. If Alice were the same height as Bob, you wouldn't say that Alice is taller than Bob or that Alice is shorter than Bob. Both of those statements would be false.

Now try entering these expressions into the interactive shell:

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
>>> 10 != 10
False
>>> 10 != 11
True
>>> 'Hello' == 'Hello'
True
>>> 'Hello' == 'Goodbye'
False
>>> 'Hello' == 'HELLO'
False
>>> 'Goodbye' != 'Hello'
True
```

The Difference Between = and ==

Try not to confuse the assignment operator (=) and the "equal to" comparison operator (==). The equal sign (=) is used in assignment statements to store a value to a variable, while the equal-equal sign (==) is used in expressions to see whether two values are equal. It's easy to accidentally use one when you meant to use the other.

Just remember that the "equal to" comparison operator (==) has two characters in it, just as the "not equal to" comparison operator (!=) has two characters in it.

String and integer values will never be equal to each other. For example, try entering the following into the interactive shell:

```
>>> 42 == 'Hello'
False
>>> 42 != '42'
True
```

Looping with `while` statements

The `while` statement marks the beginning of a loop. Loops can execute the same code repeatedly. When the execution reaches a `while` statement, it evaluates the condition next to the `while` keyword. If the condition evaluates to `True`, the execution moves inside the following block, called the while-block. (In the program, the while-block begins on line 13.) If the condition evaluates to `False`, the execution moves all the way past the while-block. In Guess the Number, the first line after the while-block is line 28.

A `while` statement always has a : colon after the condition. Statements that end with a colon expect a new block on the next line.

```
12. while guessesTaken < 6:
```


Figure 4-2: The `while` loop's condition.

Figure 4-2 shows how the execution flows depending on the condition. If the condition evaluates to `True` (which it does the first time, because the value of `guessesTaken` is 0), execution will

enter the while-block at line 13 and keep going down. Once the program reaches the end of the while-block, instead of going down to the next line, the execution loops back up to the `while` statement's line (line 12) and re-evaluates the condition. As before, if the condition is `True` the execution enters the while-block again. Each time the execution goes through the loop is called an **iteration**.

This is how the loop works. As long as the condition is `True`, the program keeps executing the code inside the while-block repeatedly until the first time the condition is `False`. Think of the `while` statement as saying, "while this condition is true, keep executing the code in the following block".

The Player Guesses

```
13.     print('Take a guess.') # There are four spaces in front of print.
14.     guess = input()
```

Lines 13 to 17 ask the player to guess what the secret number is and lets them enter their guess. That number is stored in a variable named `guess`.

Converting Values with the `int()`, `float()`, `str()`, and `bool()` Functions

```
15.     guess = int(guess)
```

Line 15 calls a new function named `int()`. The `int()` function takes one argument and returns an integer value form of that argument. Try entering the following into the interactive shell:

```
>>> int('42')
42
>>> 3 + int('2')
5
```

The `int('42')` call will return the integer value 42. However, even though you can pass a string to the `int()` function, you cannot pass just any string. Passing `'forty-two'` to `int()` will result in an error. The string you pass to `int()` must be made up of numbers:

```
>>> int('forty-two')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
int('forty-two')
ValueError: invalid literal for int() with base 10: 'forty-two'
```

The `3 + int('2')` line shows an expression that uses the return value of `int()` as part of an expression. It evaluates to the integer value 5:

```
3 + int('2')
   ▼
3 + 2
   ▼
   5
```

Remember, the `input()` function always returns **a string** of text the player typed. If the player types 5, the `input()` function will return the string value `'5'`, not the integer value 5. Python cannot use the `<` and `>` comparison operators to compare a string and an integer value:

```
>>> 4 < '5'
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    4 < '5'
TypeError: unorderable types: int() < str()
```

```
14.      guess = input()
15.      guess = int(guess)
```

On line 14 the `guess` variable originally held the string value of what the player typed. Line 15 overwrites the string value in `guess` with the integer value returned by `int()`. This lets the code later in the program compare if `guess` is greater than, less than, or equal to the secret number in the `number` variable.

One last thing: Calling `int(guess)` doesn't change the value in the `guess` variable. The code `int(guess)` is an expression that evaluates to the integer value form of the string stored in the `guess` variable. What changes `guess` is the assignment statement: `guess = int(guess)`

The `float()`, `str()`, and `bool()` functions will similarly return float, string, and Boolean versions of the arguments passed to them. Try entering the following into the interactive shell:

```
>>> float('42')
42.0
>>> float(42)
42.0
>>> str(42)
'42'
>>> str(42.0)
'42.0'
>>> str(False)
'False'
>>> bool('')
```

```
False
>>> bool('any nonempty string')
True
```

Using the int(), float(), str(), and bool() functions, you can take a value of one data type and return it as a value of a different data type.

Incrementing Variables

```
17.     guessesTaken = guessesTaken + 1
```

Once the player has taken a guess, the number of guesses should be increased by one.

On the first iteration of the loop, guessesTaken has the value of 0. Python will take this value and add 1 to it. 0 + 1 evaluates to 1, which is stored as the new value of guessesTaken. Think of line 17 as meaning, "the guessesTaken variable should be one more than what it already is".

Adding one to a variable's integer or float value is called **incrementing** the variable. Subtracting one from a variable's integer or float value is called **decrementing** the variable.

if **statements**

```
19.     if guess < number:
20.         print('Your guess is too low.') # There are eight spaces in front
of print.
```

Line 19 is an if statement. The execution will run the code in the following block if the if statement's condition evaluates to True. If the condition is False, then the code in the if-block is skipped. Using if statements, you can make the program only run certain code when you want it to.

Line 19 checks if the player's guess is less than the computer's secret number. If so, then the execution moves inside the if-block on line 20 and prints a message telling the player this.

The if statement works almost the same as a while statement, too. But unlike the while-block, the execution doesn't jump back to the if statement at the end of the if-block. It just continues down to the next line. In other words, if statements don't loop. See Figure 4-3 for a comparison of the two statements.

Figure 4-3: if and while statements.

```
22.      if guess > number:
23.          print('Your guess is too high.')
```

Line 22 checks if the player's guess is greater than the secret number. If this condition is True, then the print() function call tells the player that their guess is too high.

Leaving Loops Early with the break statement

```
25.      if guess == number:
26.          break
```

The if statement on line 25 checks if the guess is equal to the secret number. If it is, the program runs the break statement on line 26.

A **break statement** tells the execution to jump immediately out of the while-block to the first line after the end of the while-block. The break statement doesn't bother rechecking the while loop's condition.

The break statement is only found inside loops, such as in a while-block.

If the player's guess isn't equal to the secret number, the execution reaches the bottom of the while-block. This means the execution will loop back to the top and recheck the condition on line 12 (guessesTaken < 6). Remember after the guessesTaken = guessesTaken + 1 instruction executed, the new value of guessesTaken is 1. Because 1 < 6 is True, the execution enters the loop again.

If the player keeps guessing too low or too high, the value of guessesTaken will change to 2, then 3, then 4, then 5, then 6. When guessesTaken has the number 6 stored in it, the while

statement's condition (`guessesTaken < 6`) is `False`, since 6 isn't less than 6. Because the `while` statement's condition is `False`, the execution moves to the first line after the while-block, line 28.

Check if the Player Won

```
28. if guess == number:
```

Line 28 has no indentation, which means the while-block has ended and this is the first line after the while-block. The execution left the while-block either because the `while` statement's condition was `False` (when the player runs out of guesses) or the `break` statement on line 26 was executed (when the player guesses the number correctly).

Line 28 checks to see if the player guessed correctly. If so, the execution enters the if-block at line 29.

```
29.     guessesTaken = str(guessesTaken)
30.     print('Good job, ' + myName + '! You guessed my number in ' +
guessesTaken + ' guesses!')
```

Lines 29 and 30 only execute if the condition in the `if` statement on line 28 was `True` (that is, if the player correctly guessed the computer's number).

Line 29 calls the `str()` function, which returns the string form of `guessesTaken`. Line 30 concatenates strings to tell the player they have won and how many guesses it took them. Only string values can concatenate to other strings. This is why line 29 had to change `guessesTaken` to the string form. Otherwise, trying to concatenate a string to an integer would cause Python to display an error.

Check if the Player Lost

```
32. if guess != number:
```

Line 32 uses the "not equal to" comparison operator `!=` to check if player's last guess is not equal to the secret number. If this condition evaluates to `True`, the execution moves into the if-block on line 33.

Lines 33 and 34 are inside the if-block, and only execute if the condition on line 32 was `True`.

```
33.     number = str(number)
34.     print('Nope. The number I was thinking of was ' + number)
```

In this block, the program tells the player what the secret number they failed to guess correctly was. This requires concatenating strings, but `number` stores an integer value. Line 33 will overwrite `number` with a string form so that it can be concatenated to the `'Nope. The number I was thinking of was '` string on line 34.

At this point, the execution has reached the end of the code, and the program terminates. Congratulations! You've just programmed your first real game!

You can change the game's difficulty by changing the number of guesses the player gets. To give the player only four guesses, change the code on line 12:

```
12. while guessesTaken < 6:
```

into this line:

```
12. while guessesTaken < 4:
```

Code later in the while-block increases the `guessesTaken` variable by 1 on each iteration. By setting the condition to `guessesTaken < 4`, you ensure that the code inside the loop only runs four times instead of six. This makes the game much more difficult. To make the game easier, set the condition to `guessesTaken < 8` or `guessesTaken < 10`. This will cause the loop to run a few *more* times and accept *more* guesses from the player.

Flow Control Statements

In previous chapters, the program execution started at the top instruction in program and went straight down, executing each instruction in order. But with the `while`, `if`, `else`, and `break` statements, you can cause the execution to loop and skip instructions based on conditions. The name for these kinds of statements is **flow control statement**, since they change the "flow" of the program execution as it moves around your program.

Summary

If someone asked you, **"What exactly is programming anyway?"** what could you say to them? Programming is just the action of writing code for programs, that is, creating programs that can be executed by a computer.

"But what exactly is a program?" When you see someone using a computer program (for example, playing your "Guess the Number" game), all you see is some text appearing on the screen. The program decides what exact text to show on the screen (the program's **output**), based

on its instructions and on the text that the player typed on the keyboard (the program's **input**). A **program** is just a collection of instructions that act on the user's input.

"What kind of instructions?" There are only a few different kinds of instructions, really.

1. **Expressions** are values connected by operators. Expressions are all evaluated down to a single value, as `2 + 2` evaluates to `4` or `'Hello' + ' ' + 'World'` evaluates to `'Hello World'`. When expressions are next to the if and `while` keywords, you can also call them conditions.
2. **Assignment statements** store values in variables so you can remember the values later in the program.
3. **The `if`, `while`, and `break` statements** are flow control statements that can cause the execution to skip instructions, loop over instructions, or break out of loops. Function calls also change the flow of execution by jumping to the instructions inside of a function.
4. **The `print()` and `input()` functions.** These functions display text on the screen and get text from the keyboard. This is called **I/O** (pronounced like the letters, "eye-oh"), because it deals with the Input and Output of the program.

And that's it, just those four things. Of course, there are many details about those four types of instructions. In this book you'll learn about new data types and operators, new flow control statements, and many other functions that come with Python. There are also different types of I/O such as input from the mouse or outputting sound and graphics instead of just text.

For the person using your programs, they only care about that last type, I/O. The user types on the keyboard and then sees things on the screen or hears things from the speakers. But for the computer to figure out what sights to show and what sounds to play, it needs a program, and programs are just a bunch of instructions that you, the programmer, have written.

Chapter 5

JOKES

Topics Covered In This Chapter:
- Escape characters
- Using single quotes and double quotes for strings
- Using `print()`'s end keyword argument to skip newlines

Making the Most of `print()`

Most of the games in this book will have simple text for input and output. The input is typed by the user on the keyboard. The output is the text displayed on the screen. In Python, the `print()` function displays textual output on the screen. But there's more to learn about how strings and `print()` work in Python.

This chapter's program tells a few different jokes to the user, and demonstrates advanced string and `print()` code.

Sample Run of Jokes

```
What do you get when you cross a snowman with a vampire?
Frostbite!
What do dentists call an astronaut's cavity?
A black hole!
Knock knock.
Who's there?
Interrupting cow.
Interrupting cow wh-MOO!
```

Source Code of Jokes

Open a new file editor window by clicking on the **File ► New Window**. In the blank window that appears type in the source code and save it as *jokes.py*. Then run the program by pressing **F5**.

> **IMPORTANT NOTE!** The programs in this book will only run on Python 3, not Python 2. When the IDLE window starts, it will say something like "Python 3.4.2" at the top. If you have Python 2 installed, you can have Python 3 installed at the same time. To download Python 3, go to https://python.org/download/.

If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/jokes.

```
                                                                      jokes.py
 1. print('What do you get when you cross a snowman with a vampire?')
 2. input()
 3. print('Frostbite!')
 4. print()
 5. print('What do dentists call a astronaut\'s cavity?')
 6. input()
 7. print('A black hole!')
 8. print()
 9. print('Knock knock.')
10. input()
11. print("Who's there?")
12. input()
13. print('Interrupting cow.')
14. input()
15. print('Interrupting cow wh', end='')
16. print('-MOO!')
```

How the Code Works

```
 1. print('What do you get when you cross a snowman with a vampire?')
 2. input()
 3. print('Frostbite!')
 4. print()
```

Lines 1 to 4 have three `print()` function calls. You don't want the player to immediately read the joke's punch line, so there's a call to the `input()` function after the first `print()`. The player can read the joke, press ENTER, and then read the punch line.

The user can still type in a string and hit ENTER, but this returned string isn't being stored in any variable. The program will just forget about it and move to the next line of code.

The last `print()` function call has no string argument. This tells the program to just print a blank line. Blank lines are useful to keep the text from being bunched up.

Escape Characters

```
 5. print('What do dentists call a astronaut\'s cavity?')
 6. input()
 7. print('A black hole!')
```

```
8. print()
```

On line 5, there's a backslash right before the single quote: \'. Note that \ is a backslash, and / is a forward slash. This backslash tells you that the letter right after it is an escape character. An **escape character** lets you print characters that are hard to enter into the source code. On line 5 the escape character is the single quote.

The single quote escape character is there because otherwise Python would think the quote meant the end of the string. But this quote needs to be *a part of* the string. The escaped single quote tells Python that the single quote is literally a part of the string rather than marking the end of the string value.

Some Other Escape Characters

What if you really want to display a backslash? This instruction would not work:

```
>>> print('They flew away in a green\teal helicopter.')
They flew away in a green    eal helicopter.
```

This is because the "t" in "teal" was seen as an escape character since it came after a backslash. The escape character t simulates pushing the Tab key on your keyboard. Instead, try this line:

```
>>> print('They flew away in a green\\teal helicopter.')
They flew away in a green\teal helicopter.
```

Table 5-1 is a list of escape characters in Python.

Table 5-1: Escape Characters

Escape Character	What Is Actually Printed
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\n	Newline
\t	Tab

Quotes and Double Quotes

Strings don't always have to be between single quotes in Python. You can also put them between double quotes. These two lines print the same thing:

```
>>> print('Hello world')
Hello world
```

```
>>> print("Hello world")
Hello world
```

But you cannot mix quotes. This line will give you an error if you try to use them:

```
>>> print('Hello world")
SyntaxError: EOL while scanning single-quoted string
```

I like to use single quotes so I don't have to hold down the shift key to type them. It's easier to type, and Python doesn't care either way.

Just like you need the escape character \' to have a single quote in a string surrounded by single quotes, you need the escape character \" to have a double quote in a string surrounded by double quotes. For example, look at these two lines:

```
>>> print('I asked to borrow Abe\'s car for a week. He said, "Sure."')
I asked to borrow Abe's car for a week. He said, "Sure."

>>> print("She said, \"I can't believe you let them borrow your car.\"")
She said, "I can't believe you let them borrow your car."
```

In the single quote strings you don't need to escape double quotes, and in the double quote strings you don't need to escape single quotes: `"astronaut's"`. The Python interpreter is smart enough to know that if a string starts with one type of quote, the other type of quote doesn't mean the string is ending.

`print()`'s end Keyword Argument

```
 9. print('Knock knock.')
10. input()
11. print("Who's there?")
12. input()
13. print('Interrupting cow.')
14. input()
15. print('Interrupting cow wh', end='')
16. print('-MOO!')
```

Did you notice the second parameter on line 15's `print()`? Normally, `print()` adds a newline character to the end of the string it prints. This is why a blank `print()` function will just print a newline. But the `print()` function can optionally have a second parameter (which has the name `end`.)

The blank string passed is called a **keyword argument**. The end parameter has a specific name, and to pass a keyword argument to this specific parameter you must type end= before it.

By passing a blank string for end, the print() function won't add a newline at the end of the string, but instead add a blank string. This is why '-MOO!' appears next to the previous line, instead of on its own new line. There was no newline after the 'Interrupting cow wh' string was printed.

Summary

This chapter explores the different ways you can use the print() function. Escape characters are used for characters that are difficult or impossible to type into the code with the keyboard. Escape characters are typed into strings beginning with a backslash \ followed by a single letter for the escape character. For example, \n would be a newline. To include a backslash in a string, you would use the escape character \\.

The print() function automatically appends a newline character to the end of the string passed it to be displayed on the screen. Most of the time, this is a helpful shortcut. But sometimes you don't want a newline character at the end. To change this, you can pass the end keyword argument with a blank string. For example, to print "spam" to the screen without a newline character, you would call print('spam', end='').

Python provides many flexible ways to display text on the screen.

Chapter 6

DRAGON REALM

Functions

You've already used a few functions: `print()`, `input()`, `random.randint()`, `str()`, and `int()`. You've called these functions to execute the code inside them. In this chapter, you'll write your own functions for your programs to call. A function is like a mini-program inside a program.

Functions let you run the same code multiple times without duplicating the source code several times. Instead, you can put that code inside a function and call the function several times. This has the added benefit that if the function's code has a mistake, you only have one place in the program to change it.

The game you will create in this chapter is called "Dragon Realm". The player decides between two caves which hold either treasure or certain doom.

How to Play Dragon Realm

In this game, the player is in a land full of dragons. The dragons all live in caves with their large piles of collected treasure. Some dragons are friendly and share their treasure with you. Other dragons are hungry and eat anyone who enters their cave. The player is in front of two caves, one with a friendly dragon and the other with a hungry dragon. The player must choose between the two.

Open a new file editor window by clicking on the **File ► New Window**. In the blank window that appears type in the source code and save it as *dragon.py*. Then run the program by pressing **F5**.

Sample Run of Dragon Realm

```
You are in a land full of dragons. In front of you,
you see two caves. In one cave, the dragon is friendly
and will share his treasure with you. The other dragon
is greedy and hungry, and will eat you on sight.
Which cave will you go into? (1 or 2)
1
You approach the cave...
It is dark and spooky...
A large dragon jumps out in front of you! He opens his jaws and...
Gobbles you down in one bite!
Do you want to play again? (yes or no)
no
```

Source Code of Dragon Realm

> **IMPORTANT NOTE!** The programs in this book will only run on Python 3, not
> Python 2. When the IDLE window starts, it will say something like "Python
> 3.4.2" at the top. If you have Python 2 installed, you can have Python 3 installed
> at the same time. To download Python 3, go to https://python.org/download/.

If you get errors after typing this code in, compare the code you typed to the book's code with the
online diff tool at http://invpy.com/diff/dragon.

dragon.py
```
 1. import random
 2. import time
 3.
 4. def displayIntro():
 5.     print('You are in a land full of dragons. In front of you,')
 6.     print('you see two caves. In one cave, the dragon is friendly')
 7.     print('and will share his treasure with you. The other dragon')
 8.     print('is greedy and hungry, and will eat you on sight.')
 9.     print()
10.
11. def chooseCave():
12.     cave = ''
13.     while cave != '1' and cave != '2':
14.         print('Which cave will you go into? (1 or 2)')
15.         cave = input()
16.
17.     return cave
18.
```

```
19. def checkCave(chosenCave):
20.     print('You approach the cave...')
21.     time.sleep(2)
22.     print('It is dark and spooky...')
23.     time.sleep(2)
24.     print('A large dragon jumps out in front of you! He opens his jaws
and...')
25.     print()
26.     time.sleep(2)
27.
28.     friendlyCave = random.randint(1, 2)
29.
30.     if chosenCave == str(friendlyCave):
31.         print('Gives you his treasure!')
32.     else:
33.         print('Gobbles you down in one bite!')
34.
35. playAgain = 'yes'
36. while playAgain == 'yes' or playAgain == 'y':
37.
38.     displayIntro()
39.
40.     caveNumber = chooseCave()
41.
42.     checkCave(caveNumber)
43.
44.     print('Do you want to play again? (yes or no)')
45.     playAgain = input()
```

How the Code Works

Let's look at the source code in more detail.

```
1. import random
2. import time
```

This program imports two modules. The `random` module will provide the `random.randint()` function like it did in the "Guess the Number" game. You will also want time-related functions that the `time` module includes, so line 2 imports the `time` module.

`def` Statements

```
4. def displayIntro():
5.     print('You are in a land full of dragons. In front of you,')
```

```
6.      print('you see two caves. In one cave, the dragon is friendly')
7.      print('and will share his treasure with you. The other dragon')
8.      print('is greedy and hungry, and will eat you on sight.')
9.      print()
```

Line 4 is a def statement. The def statement **defines** a new function that you can call later in the program. When you *define* this function, you specify the instructions in its def-block. When you *call* this function, the code inside the def-block executes.

Figure 6-1 shows the parts of a def statement. It has the def keyword followed by a function name with parentheses and then a colon (the : sign). The block after the def statement is called the def-block.

Figure 6-1: Parts of a def statement.

Remember, the def statement doesn't execute the code. It only defines what code to execute when you call the function. When the execution reaches a def statement it skips down to the first line after the def-block.

But when the displayIntro() function is called (such as on line 38), the execution moves inside of the displayIntro() function to the first line of the def-block.

```
38.      displayIntro()
```

Then all of the print() calls are run and the "You are in a land full of dragons..." introduction is displayed.

Where to Put Function Definitions

A function's def statement and the def-block must come *before* you call the function. This is like how you must assign a value to a variable before you use the variable. If you put the function call before the function definition, you'll get an error. For example, look at this code:

```
sayGoodbye()

def sayGoodbye():
```

```
    print('Goodbye!')
```

If you try to run it, Python will give you an error message that looks like this:

```
Traceback (most recent call last):
  File "C:\Python34\spam.py", line 1, in <module>
sayGoodbye()
NameError: name 'sayGoodbye' is not defined
```

To fix this, put the function definition before the function call:

```
def sayGoodbye():
    print('Goodbye!')

sayGoodbye()
```

Defining the *chooseCave()* Function

```
11. def chooseCave():
```

Line 11 defines another function called `chooseCave()`. This function's code asks the player which cave they want to go in, either 1 or 2.

```
12.     cave = ''
13.     while cave != '1' and cave != '2':
```

This function needs to make sure the player typed 1 or 2, and not something else. A loop here will keep asking the player until they enter one of these two valid responses. This is called **input validation**.

Line 12 creates a new variable called `cave` and stores a blank string in it. Then a `while` loop begins on line 13. The condition contains a new operator you haven't seen before called `and`. Just like the `-` or `*` are mathematical operators, and `==` or `!=` are comparison operators, the `and` operator is a Boolean operator.

Boolean Operators

Boolean logic deals with things that are either `True` or `False`. Boolean operators compare values and evaluate to a single Boolean value.

Think of the sentence, "Cats have whiskers and dogs have tails." "Cats have whiskers" is true and "dogs have tails" is also true, so the entire sentence "Cats have whiskers **and** dogs have tails" is true.

But the sentence, "Cats have whiskers and dogs have wings" would be false. Even though "cats have whiskers" is true, dogs do not have wings, so "dogs have wings" is false. In Boolean logic, things can only be entirely true or entirely false. Because of the word "and", the entire sentence is only true if **both** parts are true. If one or both parts are false, then the entire sentence is false.

The *and* and *or* Operators

The and operator in Python is the same. If the Boolean values on both sides of the and keyword are True, then the expression evaluates to True. If either or both of the Boolean values are False, then the expression evaluates to False.

Try entering the following expressions with the and operator into the interactive shell:

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
>>> spam = 'Hello'
>>> 10 < 20 and spam == 'Hello'
True
```

The or operator is similar to the and operator, except it will evaluate to True if *either* of the two Boolean values are True. The only time the or operator evaluates to False is if *both* of the Boolean values are False.

Try entering the following into the interactive shell:

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
>>> 10 > 20 or 20 > 10
True
```

The *not* Operator

The not operator only works on one value, instead of combining two values. The not operator evaluates to the opposite Boolean value. The expression not True will evaluate to False and not False will evaluate to True.

Try entering the following into the interactive shell:

```
>>> not True
False
>>> not False
True
>>> not ('black' == 'white')
True
```

Truth Tables

If you ever forget how the Boolean operators work, you can look at these **truth tables**:

Table 6-1: The and operator's truth table.

A	and	B	is	Entire statement
True	and	True	is	True
True	and	False	is	False
False	and	True	is	False
False	and	False	is	False

Table 6-2: The or operator's truth table.

A	or	B	is	Entire statement
True	or	True	is	True
True	or	False	is	True
False	or	True	is	True
False	or	False	is	False

Table 6-3: The not operator's truth table.

not A	is	Entire statement
not True	is	False
not False	is	True

Evaluating Boolean Operators

Look at line 13 again:

```
13.       while cave != '1' and cave != '2':
```

The condition has two parts connected by the and Boolean operator. The condition is True only if both parts are True.

The first time the while statement's condition is checked, cave is set to the blank string, ''. The blank string is not equal to the string '1', so the left side evaluates to True. The blank string is also not equal to the string '2', so the right side evaluates to True.

So the condition then turns into True and True. Because both values are True, the condition finally evaluates to True. So the program execution enters the while-block.

This is what the evaluation looks like (if the value of cave is the blank string):

```
while cave != '1' and cave != '2':
                 ▼
while  ''  != '1' and cave != '2':
                 ▼
while     True     and cave != '2':
                 ▼
while     True     and  ''  != '2':
                 ▼
while     True     and     True:
                 ▼
while              True:
```

Getting the Player's Input

```
13.       while cave != '1' and cave != '2':
14.           print('Which cave will you go into? (1 or 2)')
15.           cave = input()
```

Line 14 asks the player which cave they choose. Line 15 lets the player type the response and hit ENTER. This response is stored in cave. After this code is executed, the execution loops back to the top of the while statement and rechecks the condition.

If the player typed in 1 or 2, then cave will either be '1' or '2' (since input() always returns strings). This makes the condition False, and the program execution will continue past the while loop. For example, if the user entered '1' then the evaluation would look like this:

```
while cave != '1' and cave != '2':
                   ▼
while  '1' != '1' and cave != '2':
                   ▼
while     False   and cave != '2':
                   ▼
while     False   and  '1' != '2':
                   ▼
while     False   and     True:
                   ▼
while           False:
```

But if the player typed 3 or 4 or HELLO, that response would be invalid. The condition will be True and enters the while-block to ask the player again. The program will keep asking until the player types 1 or 2. This will guarantee that once the execution moves on, the cave variable contains a valid response.

Return Values

```
17.      return cave
```

This is a return statement, which only appears inside def-blocks. Remember how the input() function returns a string value that the player typed in? The chooseCave() function will also return a value. Line 17 returns the string that is stored in cave, either '1' or '2'.

Once the return statement executes, the program execution jumps immediately out of the def-block. (This is like how the break statement will make the execution jump out of a while-block.) The program execution moves back to the line with the function call. The function call itself will evaluate to the return value.

Skip down and look at line 40 for a moment:

```
40.      caveNumber = chooseCave()
```

When the chooseCave() function is later called by the program on line 40, the return value is stored in the caveNumber variable. The while loop guarantees that chooseCave() will only return either '1' or '2' as its return value.

So when line 17 returns a string, the function call on line 40 evaluates to this string, which is then stored in caveNumber.

Global Scope and Local Scope

Your program's variables are forgotten after the program terminates. The variables created while the execution is inside a function call are the same. The variables are created when the function is called and forgotten when the function returns. Remember, functions are kind of like mini-programs in your program.

When execution is inside a function, you cannot change the variables outside of the function, including variables inside other functions. This is because these variables exist in a different "scope". All variables exist in either the global scope or a function call's local scope.

The scope outside of all functions is called the **global scope**. The scope inside of a function (for the duration of a particular function call) is called a **local scope**.

The entire program has only one global scope. Variables defined in the global scope can be read outside and inside functions, but can only be modified outside of all functions. Variables created in a function call can only be read or modified during that function call.

You can read the value of global variables from the local scope, but attempting to change a global variable from the local scope won't work. What Python actually does in that case is create a local variable with the **same name** as the global variable. You could, for example, have a local variable named spam at the same time as having a global variable named spam. Python will consider these to be two different variables.

Look at this example to see what happens when you try to change a global variable from inside a local scope. The comments explain what is going on:

```python
def bacon():
    # We create a local variable named "spam"
    # instead of changing the value of the global
    # variable "spam":
    spam = 99
    # The name "spam" now refers to the local
    # variable only for the rest of this
    # function:
    print(spam)     # 99

spam = 42 # A global variable named "spam":
print(spam) # 42
bacon() # Call the bacon() function:
# The global variable was not changed in bacon():
print(spam)     # 42
```

When run, this code will output the following:

```
42
99
42
```

Where a variable is created determines what scope it is in. When the Dragon Realm program first executes the line:

```
12.       cave = ''
```

...the variable cave is created inside the chooseCave() function. This means it is created in the chooseCave() function's local scope. It will be forgotten when chooseCave() returns, and will be recreated if chooseCave() is called a second time. The value of a local variable isn't remembered between function calls.

Parameters

```
19. def checkCave(chosenCave):
```

The next function the program defines is named checkCave(). Notice the text chosenCave between the parentheses. This is a **parameter**: a local variable that is assigned the argument passed when this function is called.

Remember how for some function calls like str() or randint(), you would pass one or more arguments between the parentheses:

```
>>> str(5)
'5'
>>> random.randint(1, 20)
14
```

You will also pass an argument when you call checkCave(). This argument is stored in a new variable named chosenCave. These variables are also called parameters.

For example, here is a short program that demonstrates defining a function with a parameter:

```
def sayHello(name):
    print('Hello, ' + name + '. Your name has ' + str(len(name)) + ' letters.')

sayHello('Alice')
sayHello('Bob')
spam = 'Carol'
sayHello(spam)
```

If you run this program, it would look like this:

```
Hello, Alice. Your name has 5 letters.
Hello, Bob. Your name has 3 letters.
Hello, Carol. Your name has 5 letters.
```

When you call `sayHello()`, the argument is assigned to the `name` parameter. Parameters are just ordinary local variables. Like all local variables, the values in parameters will be forgotten when the function call returns.

Displaying the Game Results

Back to the game's source code:

```
20.       print('You approach the cave...')
21.       time.sleep(2)
```

The `time` module has a function called `sleep()` that pauses the program. Line 21 passes the integer value 2 so that `time.sleep()` will pause the program for 2 seconds.

```
22.       print('It is dark and spooky...')
23.       time.sleep(2)
```

Here the code prints some more text and waits for another 2 seconds. These short pauses add suspense to the game, instead of displaying the text all at once. In the previous chapter's Jokes program, you called the `input()` function to pause until the player pressed the ENTER key. Here, the player doesn't have to do anything except wait a couple seconds.

```
24.       print('A large dragon jumps out in front of you! He opens his jaws
and...')
25.       print()
26.       time.sleep(2)
```

What happens next? And how does the program decide? This is explained in the next section.

Deciding Which Cave has the Friendly Dragon

```
28.       friendlyCave = random.randint(1, 2)
```

Line 28 calls the `random.randint()` function which will return either 1 or 2. This integer value is stored in `friendlyCave` and is the cave with the friendly dragon.

```
30.      if chosenCave == str(friendlyCave):
31.          print('Gives you his treasure!')
```

Line 30 checks if the player's chosen cave in the chosenCave variable ('1' or '2') is equal to the friendly dragon cave.

But the value in friendlyCave is an integer because random.randint() returns integers. You can't compare strings and integers with the == sign, because they will **always** be not equal to each other. '1' is not equal to 1 and '2' is not equal to 2.

So friendlyCave is passed to str() function, which returns the string value of friendlyCave. This way the values will be the same data type and can be meaningfully compared to each other. This code could also have been used to convert chosenCave to an integer value:

```
    if int(chosenCave) == friendlyCave:
```

If the condition is True, line 31 tells the player they have won the treasure.

```
32.      else:
33.          print('Gobbles you down in one bite!')
```

Line 32 is an **else** statement. The else statement can only come after an if-block. The else-block executes if the if statement's condition was False. Think of it as the program's way of saying, "If this condition is true then execute the if-block or else execute the else-block."

Remember to put the colon (the : sign) after the else keyword.

Where the Main Part Begins

```
35. playAgain = 'yes'
36. while playAgain == 'yes' or playAgain == 'y':
```

Line 35 is the first line that isn't a def statement or inside a def-block. This line is where the main part of the program begins. The previous def statements merely defined the functions. They didn't run the code inside of the functions.

Line 35 and 36 are setting up a loop that the rest of the game code is in. At the end of the game, the player can enter if they want to play again. If they do, the execution enters the while loop to run the entire game all over again. If they don't, the while statement's condition will be False and the execution will move on to the end of the program and terminate.

The first time the execution comes to this `while` statement, line 35 will have just assigned `'yes'` to the `playAgain` variable. That means the condition will be `True`. This guarantees that the execution enters the loop at least once.

Calling the Functions in the Program

```
38.       displayIntro()
```

Line 38 calls the `displayIntro()` function. This isn't a Python function, it is your function that you defined earlier on line 4. When this function is called, the program execution jumps to the first line in the `displayIntro()` function on line 5. When all the lines in the function are done, the execution jumps back to line 38 and continues moving down.

```
40.       caveNumber = chooseCave()
```

Line 40 also calls a function that you defined. Remember that the `chooseCave()` function lets the player type in the cave they want to go into. When the line 17's `return cave` executes, the program execution jumps back to line 40, and the `chooseCave()` call evaluates to the return value. This return value is stored in a new variable named `caveNumber`. Then the program execution moves on to line 42.

```
42.       checkCave(caveNumber)
```

Line 42 calls your `checkCave()` function, passing the value in `caveNumber` as an argument. Not only does execution jump to line 20, but the value in `caveNumber` is copied to the parameter `chosenCave` inside the `checkCave()` function. This is the function that will display either `'Gives you his treasure!'` or `'Gobbles you down in one bite!'` depending on the cave the player chose to go into.

Asking the Player to Play Again

```
44.       print('Do you want to play again? (yes or no)')
45.       playAgain = input()
```

Whether the player won or lost, they are asked if they want to play again. The variable `playAgain` stores what the player typed. Line 45 is the last line of the while-block, so the program jumps back to line 36 to check the `while` loop's condition: `playAgain == 'yes'` or `playAgain == 'y'`

If the player typed in the string `'yes'` or `'y'`, then the execution would enter the loop again at line 38.

If the player typed in `'no'` or `'n'` or something silly like `'Abraham Lincoln'`, then the condition would be `False`. The program execution would continue on to the line after the while-block. But since there are no more lines after the while-block, the program terminates.

One thing to note: the string `'YES'` is not equal to the string `'yes'`. If the player typed in the string `'YES'`, then the `while` statement's condition would evaluate to `False` and the program would still terminate. Later programs in this book will show you how to avoid this problem.

You've just completed your second game! In Dragon Realm, you used a lot of what you learned in the Guess the Number game and picked up a few new tricks. If you didn't understand some of the concepts in this program, then go over each line of the source code again, and try changing the source code and see how the program changes.

In the next chapter you won't create a game, but instead learn how to use a feature of IDLE called the debugger.

Designing the Program

Dragon Realm is a simple game. The other games in this book will be a bit more complicated. It sometimes helps to write down everything you want your game or program to do before you start writing code. This is called "designing the program."

For example, it may help to draw a flow chart. A **flow chart** is a picture that shows every possible action that can happen in the game, and which actions lead to which other actions. Figure 6-2 is a flow chart for Dragon Realm.

To see what happens in the game, put your finger on the "Start" box. Then follow one arrow from the box to another box. Your finger is like the program execution. The program terminates when your finger lands on the "End" box.

When you get to the "Check for friendly or hungry dragon" box, you can go to the "Player wins" box or the "Player loses" box. This branching point shows how the program can do different things. Either way, both paths will end up at the "Ask to play again" box.

Summary

In the Dragon Realm game, you created your own functions. Functions are a mini-program within your program. The code inside the function runs when the function is called. By breaking up your code into functions, you can organize your code into smaller and easier to understand sections.

Arguments are values copied to the function's parameters when the function is called. The function call itself evaluates to the return value.
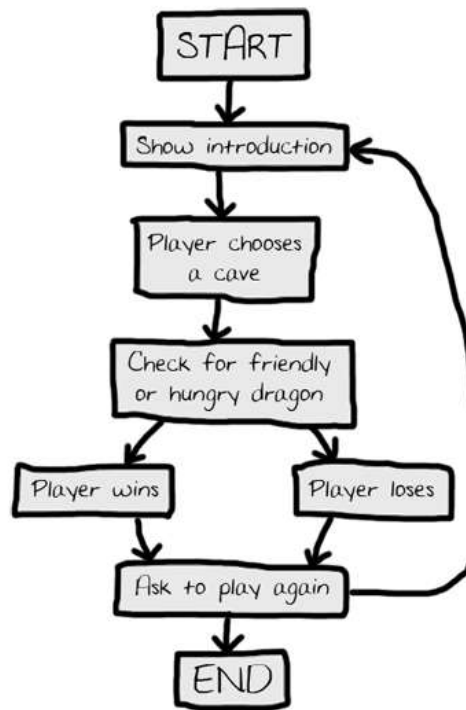
Figure 6-2: Flow chart for the Dragon Realm game.

You also learned about variable scopes. Variables created inside of a function exist in the local scope, and variables created outside of all functions exist in the global scope. Code in the global scope cannot make use of local variables. If a local variable has the same name as a variable in the global scope, Python considers it a separate variable and assigning new values to the local variable won't change the value in the global variable.

Variable scopes might seem complicated, but they are useful for organizing functions as separate pieces of code from the rest of the program. Because each function has its own local scope, you can be sure that the code in one function won't cause bugs in other functions.

Almost every program uses functions because they are so useful. By understanding how functions work, you can save yourself a lot of typing and make bugs easier to fix.

Chapter 7

USING THE DEBUGGER

Topics Covered In This Chapter:
- 3 Different Types of Errors
- IDLE's Debugger
- Stepping Into, Over, and Out
- Go and Quit
- Break Points

Bugs!

"On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

-*Charles Babbage, 19th century originator the concept of a programmable computer.*

If you enter the wrong code, the computer won't give you the right program. A computer program will always do what you tell it to, but what you tell the program to do might not be the same as what you *wanted* the program to do. These errors are **bugs** in a computer program. Bugs happen when the programmer has not carefully thought about what exactly the program is doing. There are three types of bugs that can happen with your program:

- **Syntax Errors** are a type of bug that comes from typos. When the Python interpreter sees a syntax error, it is because your code isn't written in proper Python language. A Python program with even a single syntax error won't run.
- **Runtime Errors** are bugs that happen while the program is running. The program will work up until it reaches the line of code with the error, and then the program terminates with an error message (this is called **crashing**). The Python interpreter will display a "traceback" and show the line where the problem happens.
- **Semantic Errors** are the trickiest to fix. These bugs don't crash the program, but it isn't doing what the programmer intended for the program to do. For example, if the programmer wants the variable `total` to be the *sum* of the values in variables `a`, `b`, and `c` but writes `total = a * b * c`, then the value in total will be wrong. This could crash the program later on, but it is not immediately obvious where the semantic bug happened.

Finding bugs in a program can be hard, if you even notice them at all! When running your program, you may discover that sometimes functions are not called when they are supposed to be, or maybe they are called too many times. You may code the condition for a `while` loop wrong, so that it loops the wrong number of times. (A loop in your program that never exits is a kind of bug called an **infinite loop**. To stop this program, you can press **Ctrl-C** in the interactive shell to terminate the program.) Any of these things could mistakenly happen in your code if you are not careful.

In fact, from the interactive shell, go ahead and create an infinite loop by typing this code in (you have to press ENTER twice to let the interactive shell know you are done typing in the while-block:

```
>>> while True:
...       print('Press Ctrl-C to stop this infinite loop!!!')
...
```

Now press and hold down the Ctrl key and press the C key to stop the program. The interactive shell will look like this:

```
Press Ctrl-C to stop this infinite loop!!!
Press Ctrl-C to stop this infinite loop!!!
Press Ctrl-C to stop this infinite loop!!!
Press Ctrl-C to stop this infinite loop!!!
Press Ctrl-C to stop this infinite loop!!!
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    while True: print('Press Ctrl-C to stop this infinite loop!!!')
KeyboardInterrupt
```

The Debugger

It can be hard to figure out how your code could be causing a bug. The lines of code get executed quickly and the values in variables change so often. A **debugger** is a program that lets you step through your code one line at a time in the same order that Python executes them. The debugger also shows you what values are stored in variables at each step.

Starting the Debugger

In IDLE, open the Dragon Realm game you made in the last chapter. After opening the *dragon.py* file, click on the **Debug ► Debugger** to make the Debug Control window appear (Figure 7-1).

Figure 7-1: The Debug Control window.
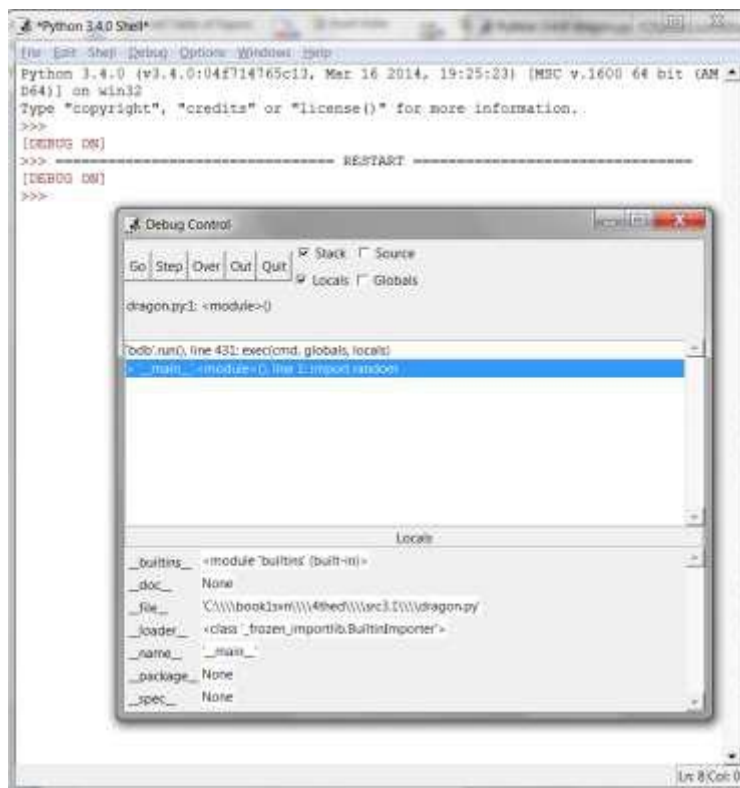
Figure 7-2: Running the Dragon Realm game under the debugger.

Now when you run the Dragon Realm game by pressing **F5**, IDLE's debugger will activate. This is called running a program "under a debugger". In the Debug Control window, check the **Source** and **Globals** checkboxes.

When you run Python programs under the debugger, the program will stop before it executes the first instruction. If you click on the file editor window's title bar (and you've checked the **Source** checkbox in the Debug Control window), the first instruction is highlighted in gray. The Debug Control window shows the execution is on line 1, which is the `import random` line.

Stepping

The debugger lets you execute one instruction at a time. This is called **stepping**. To execute a single instruction, click the **Step** button in the Debug Window. Go ahead and do this now. Python will execute the `import random` instruction, and then stop before it executes the next instruction. The Debug Control window will show the execution is now on line 2, the `import time` line. Click the **Quit** button to terminate the program for now.

Here is a summary of what happens when you click the Step button when you run the Dragon Realm game under a debugger. Press **F5** to start running Dragon Realm again, then follow these instructions:

1. Click the **Step** button twice to run the two `import` lines.
2. Click the **Step** button three more times to execute the three `def` statements.
3. Click the **Step** button again to define the `playAgain` variable.
4. Click **Go** to run the rest of the program, or click **Quit** to terminate the program.

The Debug Control window will show you what line is *about* to be executed when you click the Step button in the Debug Control window. The debugger skipped line 3 because it's a blank line. Notice you can only step forward with the debugger, you cannot go backwards.

Globals Area

The **Globals area** in the Debug Control window is where all the global variables can be seen. Remember, global variables are the variables that are created outside of any functions (that is, in the global scope).

As the three `def` statements execute and define functions, they will appear in the Globals area of the Debug Control window.

The text next to the function names in the Globals area will look like "<function checkCave at 0x012859B0>". The module names also have confusing looking text next to them, such as "<module 'random' from 'C:\\Python31\\lib\\random.pyc'>". You don't need to know what it

means to debug your programs. Just seeing that the functions and modules are there in the Global area will tell you if the function has been defined or the module has been imported.

You can also ignore the __builtins__, __doc__, and __name__ lines in the Global area. (Those are variables that appear in every Python program.)

When the playAgain variable is created it will show up in the Global area. Next to the variable name will be the string 'yes'. The debugger lets you see the values of all the variables in the program as the program runs. This is useful for fixing bugs.

Locals Area

There is also a **Locals area**, which shows you the local scope variables and their values. The local area will only have variables in it when the program execution is inside of a function. When the execution is in the global scope, this area is blank.

The Go and Quit Buttons

If you get tired of clicking the **Step** button repeatedly and just want the program to run normally, click the **Go** button at the top of the Debug Control window. This will tell the program to run normally instead of stepping.

To terminate the program entirely, just click the **Quit** button at the top of the Debug Control window. The program will exit immediately. This is helpful if you must start debugging again from the beginning of the program.

Stepping Into, Over, and Out

Start the Dragon Realm program with the debugger. Keep stepping until the debugger is at line 38. As shown in Figure 7-3, this is the line with displayIntro(). When you click **Step** again, the debugger will jump into this function call and appear on line 5, the first line in the displayIntro() function. The kind of stepping you have been doing is called **stepping into**. This is different from stepping over, explained next.
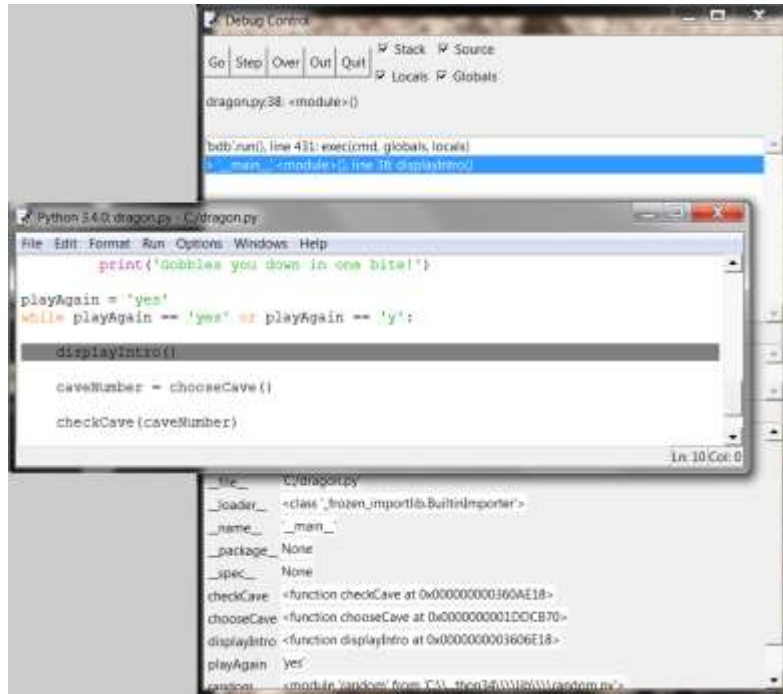
Figure 7-3: Keep stepping until you reach line 38.

When the execution is paused at line 5, clicking **Step** one more time will step into the print() function. The print() function is one of Python's built-in functions, so it isn't useful to step through with the debugger. Python's own functions such as print(), input(), str(), or random.randint() have been carefully checked for errors. You can assume they're not the parts causing bugs in your program.

So you don't want to waste time stepping through the internals of the print() function. So instead of clicking **Step** to step into the print() function's code, click **Over**. This will step *over* the code inside the print() function. The code inside print() will be executed at normal speed, and then the debugger will pause once the execution returns from print().

Stepping over is a convenient way to skip stepping through code inside a function. The debugger will now be paused at line 40, caveNumber = chooseCave().

Click **Step** one more time to step into the chooseCave() function. Keep stepping through the code until line 15, the input() call. The program will wait until you type a response into the interactive shell, just like when you run the program normally. If you try clicking the **Step** button now, nothing will happen because the program is waiting for a keyboard response.

Click back on the interactive shell window and type which cave you want to enter. The blinking cursor must be on the bottom line in the interactive shell before you can type. Otherwise the text you type will not appear.

Once you press ENTER, the debugger will continue to step lines of code again. Click the **Out** button on the Debug Control window. This is called **stepping out**, because it will cause the debugger to step over as many lines as it needs to until execution has returned from the function it is in. After it jumps out, the execution will be on the line after the line that called the function.

For example, clicking **Out** inside the `displayIntro()` function on line 6 would step until the function returned to the line after the call to `displayIntro()`. Stepping out can save you from having to click **Step** repeatedly to jump out of the function.

If you are not inside a function, clicking **Out** will cause the debugger will execute all the remaining lines in the program. This is the same behavior as clicking the **Go** button.

Here's a recap of what each button does:

- **Go** - Executes the rest of the code as normal, or until it reaches a break point. (Break points are described later.)
- **Step** - Step one instruction. If the line is a function call, the debugger will step into the function.
- **Over** - Step one instruction. If the line is a function call, the debugger won't *step into* the function, but instead *step over* the call.
- **Out** - Keeps stepping over lines of code until the debugger leaves the function it was in when **Out** was clicked. This *steps out* of the function.
- **Quit** - Immediately terminates the program.

Find the Bug

The debugger can help you find the cause of bugs in your program. As an example, here is a small program with a bug. The program comes up with a random addition problem for the user to solve. In the interactive shell window, click on File, then New Window to open a new file editor window. Type this program into that window, and save the program as *buggy.py*.

```
                                                                  buggy.py
1. import random
2. number1 = random.randint(1, 10)
3. number2 = random.randint(1, 10)
4. print('What is ' + str(number1) + ' + ' + str(number2) + '?')
5. answer = input()
6. if answer == number1 + number2:
```

```
7.     print('Correct!')
8. else:
9.     print('Nope! The answer is ' + str(number1 + number2))
```

Type the program as it is above, even if you can already tell what the bug is. Then trying running the program by pressing **F5**. This is a simple arithmetic quiz that comes up with two random numbers and asks you to add them. Here's what it might look like when you run the program:

```
What is 5 + 1?
6
Nope! The answer is 6
```

That's a bug! The program doesn't crash but it is not working correctly. The program says the user is wrong even if they type the correct answer.

Running the program under a debugger will help find the bug's cause. At the top of the interactive shell window, click on **Debug ► Debugger** to display the Debug Control window. In the Debug Control window, check all four checkboxes (Stack, Source, Locals, and Globals). This makes the Debug Control window provide the most information. Then press **F5** in the file editor window to run the program. This time it will be run under the debugger.

```
1. import random
```

The debugger starts at the import random line. Nothing special happens here, so just click **Step** to execute it. You will see the random module added to the Globals area.

```
2. number1 = random.randint(1, 10)
```

Click **Step** again to run line 2. A new file editor window will appear with the *random.py* file. You have stepped inside the randint() function inside the random module. Python's built-in functions won't be the source of your bugs, so click **Out** to step out of the randint() function and back to your program. Then close the *random.py* file's window.

```
3. number2 = random.randint(1, 10)
```

Next time, you can click **Over** to step over the randint() function instead of stepping into it. Line 3 is also a randint() function call. Skip stepping into this code by clicking **Over**.

```
4. print('What is ' + str(number1) + ' + ' + str(number2) + '?')
```

Line 4 is a `print()` call to show the player the random numbers. You know what numbers the program will print even before it prints them! Just look at the Globals area of the Debug Control window. You can see the `number1` and `number2` variables, and next to them are the integer values stored in those variables.

The `number1` variable has the value 4 and the `number2` variable has the value 8. When you click **Step**, the program will display the string in the `print()` call with these values. The `str()` function will concatenate the string version of these integers. When I ran the debugger, it looked like Figure 7-4. (Your random numbers will probably be different.)

Figure 7-4: number1 is set to 4 and number2 is set to 8.

```
5. answer = input()
```

Clicking on **Step** from line 5 will execute `input()`. The debugger waits until the player enters a response into the program. Enter the correct answer (in my case, 12) into the interactive shell window. The debugger will resume and move down to line 6.

```
6. if answer == number1 + number2:
7.     print('Correct!')
```

Line 6 is an `if` statement. The condition is that the value in `answer` must match the sum of `number1` and `number2`. If the condition is `True`, then the debugger will move to line 7. If the

condition is `False`, the debugger will move to line 9. Click `Step` one more time to find out where it goes.

```
8. else:
9.     print('Nope! The answer is ' + str(number1 + number2))
```

The debugger is now on line 9! What happened? The condition in the `if` statement must have been `False`. Take a look at the values for `number1`, `number2`, and `answer`. Notice that `number1` and `number2` are integers, so their sum would have also been an integer. But `answer` is a string.

That means that `answer == number1 + number2` would have evaluated to `'12' == 12`. A string value and an integer value will always not equal each other, so the condition evaluated to `False`.

That is the bug in the program. The bug is that the code has `answer` when it should have `int(answer)`. Change line 6 to `int(answer) == number1 + number2`, and run the program again.

```
What is 2 + 3?
5
Correct!
```

This time, the program worked correctly. Run it one more time and enter a wrong answer on purpose. This will completely test the program. You've now debugged this program! Remember, the computer will run your programs exactly as you type them, even if what you type isn't what you intend.

Break Points

Stepping through the code one line at a time might still be too slow. Often you'll want the program to run at normal speed until it reaches a certain line. A **break point** is set on a line when you want the debugger to take control once execution reaches that line. If you think there's a problem with your code on, say, line 17, just set a break point on line 17 (or maybe a few lines before that).

When execution reaches that line, the debugger will "break into the debugger". Then you can step through lines one at a time to see what is happening. Clicking **Go** will execute the program normally until it reaches another break point or the end of the program.

To set a break point, right-click on the line in the file editor and select **Set Breakpoint** from the menu that appears. The file editor will highlight that line with yellow. You can set break points on as many lines as you want. To remove the break point, click on the line and select **Clear Breakpoint** from the menu that appears.
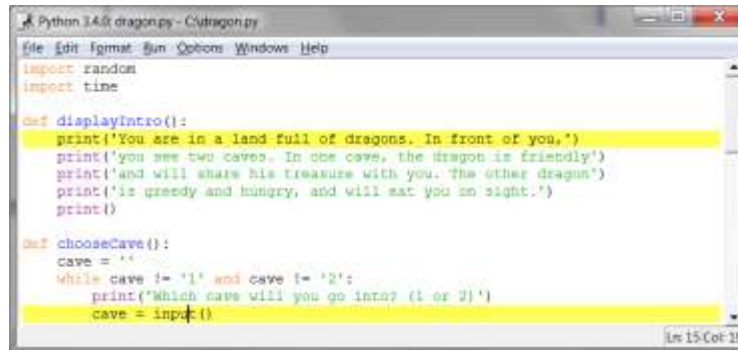
Figure 7-5: The file editor with two break points set.

Example Using Break Points

Here is a program that simulates coin flips by calling `random.randint(0, 1)`. The function returning the integer 1 will be "heads" and returning the integer 0 will be "tails". The `flips` variable will track how many coin flips have been done. The `heads` variable will track how many came up heads.

The program will do "coin flips" one thousand times. This would take a person over an hour to do, but the computer can do it in one second! Type in the following code into the file editor and save it as *coinFlips.py*. If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/coinflips.

```
                                                         coinFlips.py
 1. import random
 2. print('I will flip a coin 1000 times. Guess how many times it will come up
heads. (Press enter to begin)')
 3. input()
 4. flips = 0
 5. heads = 0
 6. while flips < 1000:
 7.     if random.randint(0, 1) == 1:
 8.         heads = heads + 1
 9.     flips = flips + 1
10.
11.     if flips == 900:
12.         print('900 flips and there have been ' + str(heads) + ' heads.')
13.     if flips == 100:
14.         print('At 100 tosses, heads has come up ' + str(heads) + ' times so
far.')
15.     if flips == 500:
```

```
16.         print('Half way done, and heads has come up ' + str(heads) + '
times.')
17.
18. print()
19. print('Out of 1000 coin tosses, heads came up ' + str(heads) + ' times!')
20. print('Were you close?')
```

The program runs pretty fast. It spent more time waiting for the user to press ENTER than doing the coin flips. Let's say you wanted to see it do coin flips one by one. On the interactive shell's window, click on **Debug ► Debugger** to bring up the Debug Control window. Then press **F5** to run the program.

The program starts in the debugger on line 1. Press **Step** three times in the Debug Control window to execute the first three lines (that is, lines 1, 2, and 3). You'll notice the buttons become disabled because input() was called and the interactive shell window is waiting for the user to type something. Click on the interactive shell window and press ENTER. (Be sure to click beneath the text in the interactive shell window, otherwise IDLE might not receive your keystrokes.)

You can click **Step** a few more times, but you'll find that it would take quite a while to get through the entire program. Instead, set a break point on lines 12, 14, and 16. The file editor will highlight these lines as shown in Figure 7-6.
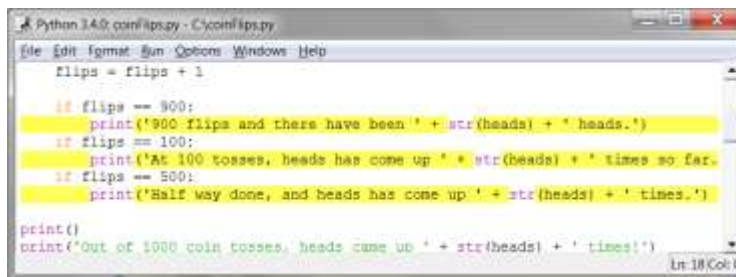
Figure 7-6: Three break points set.

After setting the breakpoints, click **Go** in the Debug Control window. The program will run at normal speed until it reaches the next break point. When flip is set to 100, the condition for the if statement on line 13 is True. This causes line 14 (where there's a break point set) to execute, which tells the debugger to stop the program and take over. Look at the Debug Control window in the Globals section to see what the value of flips and heads are.

Click **Go** again and the program will continue until it reaches the next break point on line 16. Again, see how the values in flips and heads have changed.

If you click **Go** again, the execution will continue until the next break point is reached, which is on line 12.

Summary

Writing programs is only the first part of programming. The next part is making sure the code you wrote actually works. Debuggers let you step through the code one line at a time. You can examine which lines execute in what order, and what values the variables contain. When this is too slow, you can set break points to stop the debugger only at the lines you want.

Using the debugger is a great way to understand what a program is doing. While this book provides explanations of all the game code in it, the debugger can help you find out more on your own.

Chapter 8

FLOW CHARTS

> Topics Covered In This Chapter:
> - How to play Hangman
> - ASCII art
> - Designing a program with flow charts

In this chapter, you'll design a Hangman game. This game is more complicated than our previous game, but also more fun. Because the game is advanced, you should first carefully plan it out by creating a flow chart (explained later). In the next chapter, you'll actually write out the code for Hangman.

How to Play Hangman

Hangman is a game for two people usually played with paper and pencil. One player thinks of a word, and then draws a blank on the page for each letter in the word. Then the second player tries to guess letters that might be in the word.

If they guess correctly, the first player writes the letter in the proper blank. If they guess incorrectly, the first player draws a single body part of the hanging man. If the second player can guess all the letters in the word before the hangman is completely drawn, they win. But if they can't figure it out in time, they lose.

Sample Run of Hangman

Here is an example of what the player might see when they run the Hangman program you'll write in the next chapter. The text that the player enters in shown in bold.

```
H A N G M A N
  +---+
  |   |
      |
      |
      |
      |
=========
Missed letters:
_ _ _
```

```
Guess a letter.
a
  +---+
  |   |
      |
      |
      |
      |
=========
Missed letters:
_ a _
Guess a letter.
o
  +---+
  |   |
  O   |
      |
      |
      |
=========
Missed letters: o
_ a _
Guess a letter.
r
  +---+
  |   |
  O   |
  |   |
      |
      |
=========
Missed letters: or
_ a _
Guess a letter.
t
  +---+
  |   |
  O   |
  |   |
      |
      |
=========
Missed letters: or
_ a t
Guess a letter.
a
You have already guessed that letter. Choose again.
```

```
Guess a letter.
c
Yes! The secret word is "cat"! You have won!
Do you want to play again? (yes or no)
no
```

ASCII Art

The graphics for Hangman are keyboard characters printed on the screen. This type of graphics is called **ASCII art** (pronounced "ask-ee"), which was a sort of precursor to emojii. Here is a cat drawn in ASCII art:
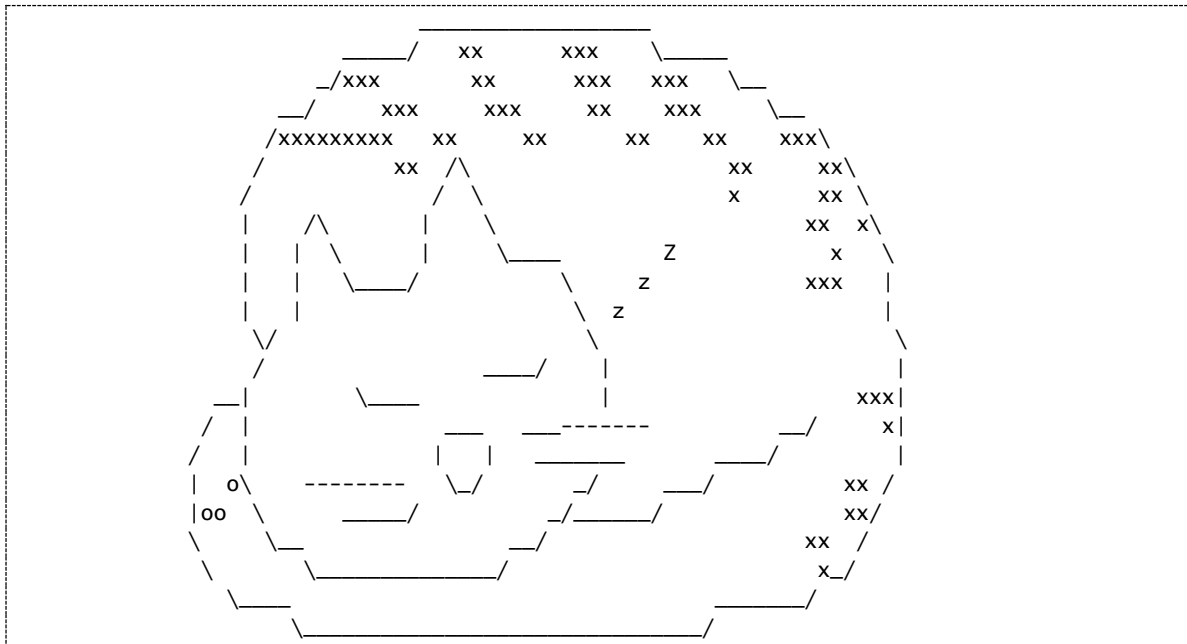
Designing a Program with a Flowchart

This game is a bit more complicated than the ones you've seen so far, so take a moment to think about how it's put together. First you'll create a flow chart (like the one at the end of the Dragon Realm chapter) to help visualize what this program will do. This chapter will go over what flow charts are and why they are useful. The next chapter will go over the source code to the Hangman game.

A **flow chart** is a diagram that shows a series of steps as boxes connected with arrows. Each box represents a step, and the arrows show the steps leads to which other steps. Put your finger on the

"Start" box of the flow chart and trace through the program by following the arrows to other boxes until you get to the "End" box.

Figure 8-1 is a complete flow chart for Hangman. You can only move from one box to another in the direction of the arrow. You can never go backwards unless there's a second arrow going back, like in the "Player already guessed this letter" box.
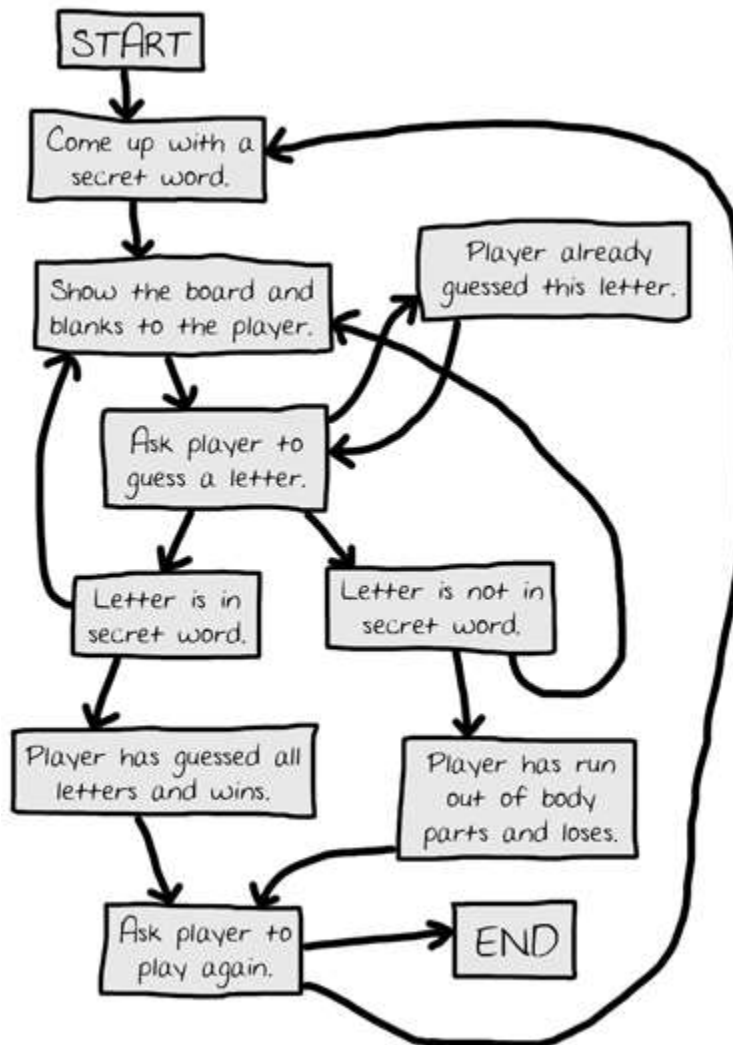
Figure 8-1: The complete flow chart for what happens in the Hangman game.

Of course, you don't *have* to make a flow chart. You could just start writing code. But often once you start programming you'll think of things that must be added or changed. You may end up

having to delete a lot of your code, which would be a waste of effort. To avoid this, it's always best to plan how the program will work before you start writing it.

Creating the Flow Chart

Your flow charts don't always have to look like this one. As long as **you** understand the flow chart you made, it will be helpful when you start coding. A flow chart that begins with just a "Start" and an "End" box, as shown in Figure 8-2:

Figure 8-2: Begin your flow chart with a Start and End box.

Now think about what happens when you play Hangman. First, the computer thinks of a secret word. Then the player will guess letters. Add boxes for these events, as shown in Figure 8-3. The new boxes in each flow chart have a dashed outline around them.

The arrows show the order that the program should move. That is, first the program should come up with a secret word, and after that it should ask the player to guess a letter.
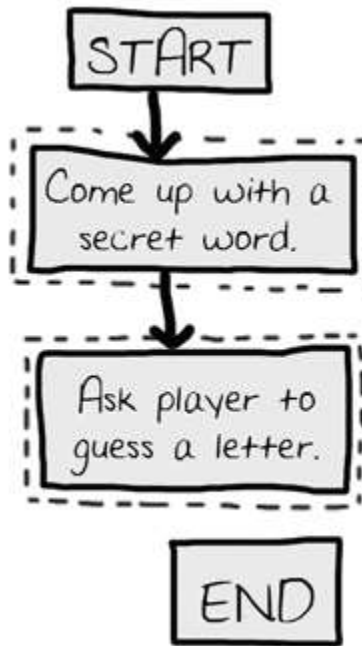
Figure 8-3: Draw out the first two steps of Hangman as boxes with descriptions.

But the game doesn't end after the player guesses one letter. It needs to check if that letter is in the secret word or not.

Branching from a Flowchart Box

There are two possibilities: the letter is either in the word or not. You'll add two new boxes to the flowchart, one for each case. This creates a branch in the flow chart, as show in Figure 8-4:
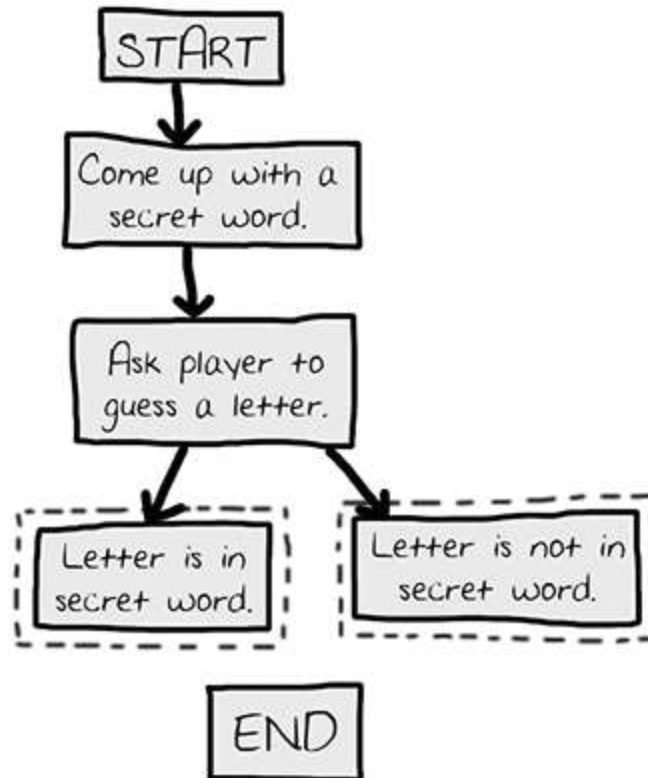
Figure 8-4: The branch has two arrows going to separate boxes.

If the letter is in the secret word, check if the player has guessed all the letters and won the game. If the letter isn't in the secret word, another body part is added to the hanging man. Add boxes for those cases too.

You **don't** need an arrow from the "Letter is in secret word" box to the "Player has run out of body parts and loses" box, because it's impossible to lose as long as the player guesses correctly. It's also impossible to win as long as the player is guessing incorrectly, so you don't need to draw that arrow either. The flow chart now looks like Figure 8-5.
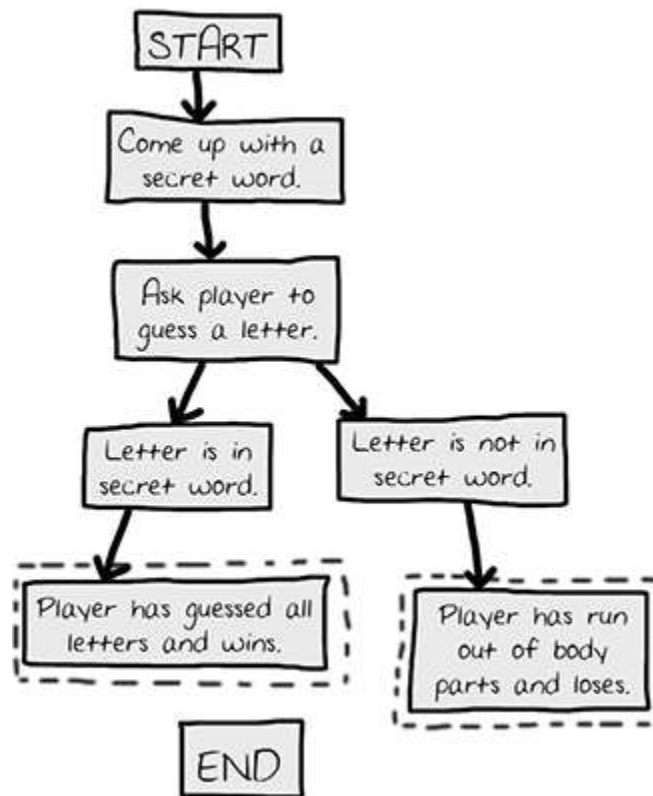
Figure 8-5: After the branch, the steps continue on their separate paths.

Ending or Restarting the Game

Once the player has won or lost, ask them if they want to play again with a new secret word. If the player doesn't want to play again, the program will end. If the program doesn't end, it thinks up a new secret word. This is shown in Figure 8-6.
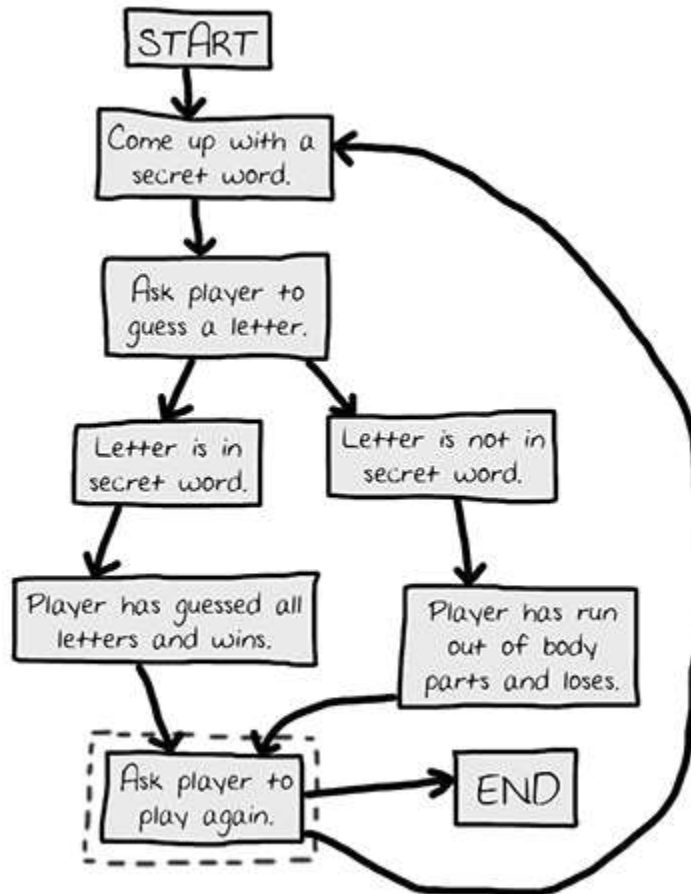
Figure 8-6: The flow chart branches when asking the player to play again.

Guessing Again

The player doesn't guess a letter just once. They have to keep guessing letters until they win or lose. You'll draw two new arrows, as shown in Figure 8-7.
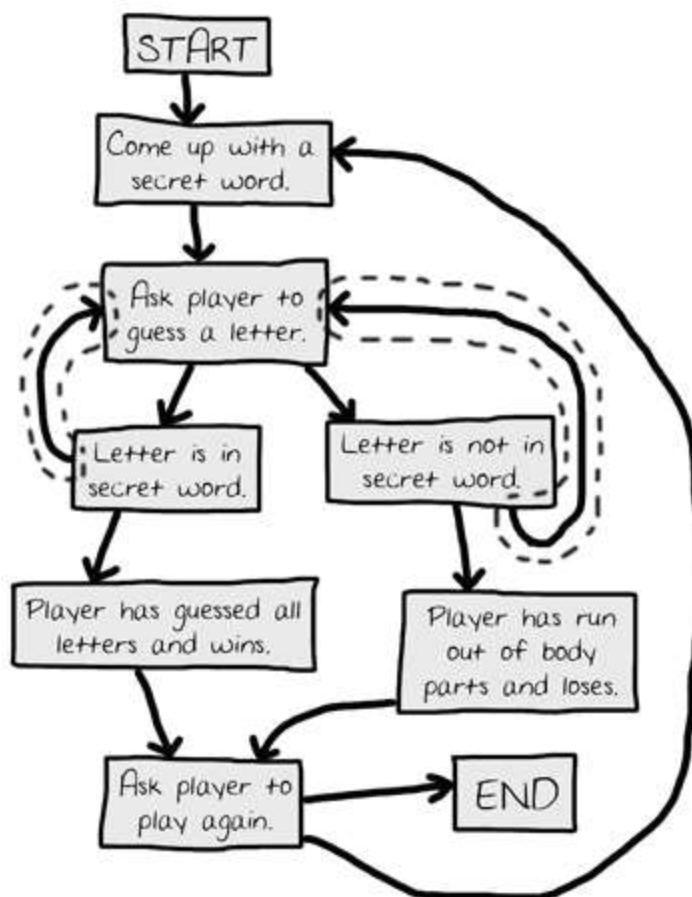
Figure 8-7: The new arrows (outlined) show the player can guess again.

What if the player guesses the same letter again? Rather than have them win or lose in this case, allow them to guess a different letter instead. This new box is shown in Figure 8-8.
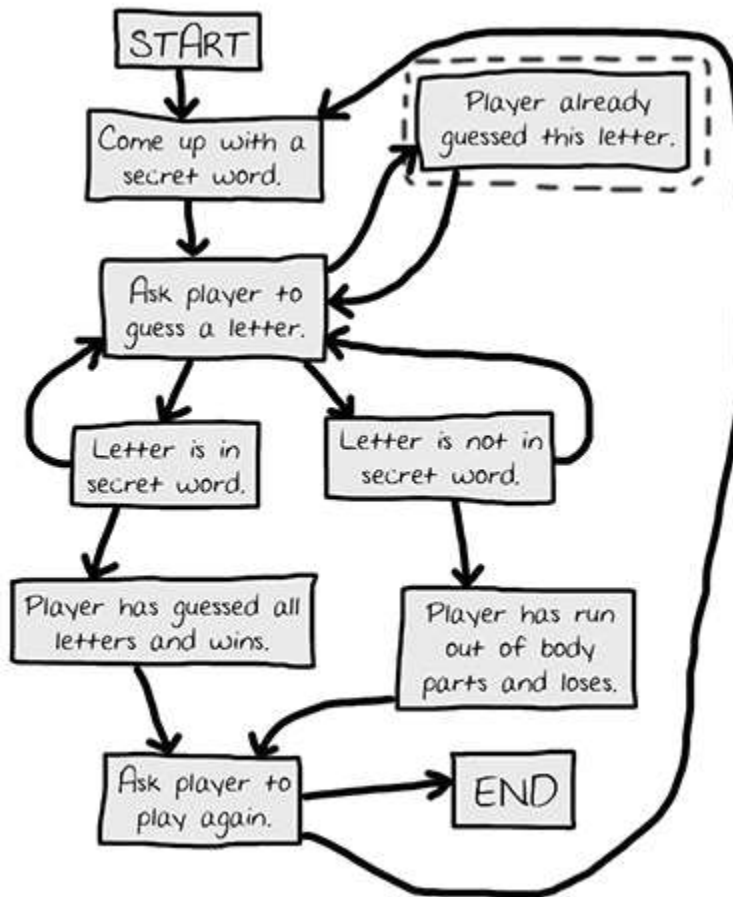
Figure 8-8: Adding a step in case the player guesses a letter they already guessed.

Offering Feedback to the Player

The player needs to know how they're doing in the game. The program should show them the hangman board and the secret word (with blanks for the letters they haven't guessed yet). These visuals will let them see how close they are to winning or losing the game.

This information is updated every time the player guesses a letter. Add a "Show the board and blanks to the player." box to the flow chart between the "Come up with a secret word" and the "Ask player to guess a letter" boxes. These boxes are shown in Figure 8-9.
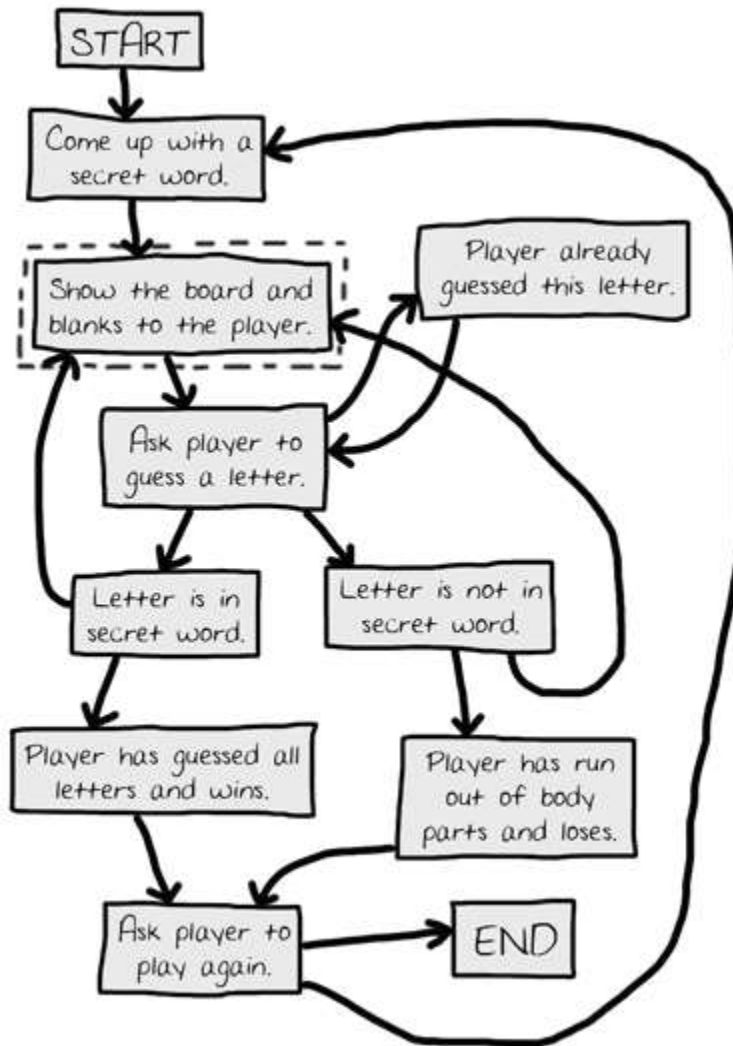
Figure 8-9: Adding "Show the board and blanks to the player." to give the player feedback.

That looks good! This flow chart completely maps out everything that can happen in Hangman and in what order. When you design your own games, a flow chart can help you remember everything you need to code.

Summary

It may seem like a lot of work to sketch out a flow chart about the program first. After all, people want to play games, not look at flowcharts! But it is much easier to make changes and notice problems by thinking about how the program works before writing the code for it.

If you jump in to write the code first, you may discover problems that require you to change the code you've already written. Every time you change your code, you are taking a chance you create new bugs by changing too little or too much. It is much better to know what you want to build before you build it.

Chapter 9

HANGMAN

> Topics Covered In This Chapter:
> - Multi-line Strings
> - Methods
> - Lists
> - The `append()` and `reverse()` list methods
> - The `lower()`, `upper()`, `split()`, `startswith()`, and `endswith()` string methods
> - The `in` and `not in` operators
> - The `range()` and `list()` functions
> - `del` statements
> - `for` loops
> - `elif` statements

This chapter's game introduces many new concepts, but don't worry. You'll experiment with these programming concepts in the interactive shell first. You'll learn about methods, which are functions attached to values. You'll also learn about a new type of loop called a `for` loop and a new data type called a list. Once you understand these concepts, it will be much easier to program Hangman.

Source Code of Hangman

This chapter's game is a bit longer than the previous games, but much of it is the ASCII art for the hangman pictures. Enter the following into the file editor and save it as *hangman.py*.

```
                                                          hangman.py
 1. import random
 2. HANGMANPICS = ['''
 3.
 4.   +---+
 5.   |   |
 6.       |
 7.       |
 8.       |
 9.       |
10. =========''', '''
11.
12.   +---+
```

```
13.    |   |
14.    O   |
15.        |
16.        |
17.        |
18. ========''', '''
19.
20.    +---+
21.    |   |
22.    O   |
23.    |   |
24.        |
25.        |
26. ========''', '''
27.
28.    +---+
29.    |   |
30.    O   |
31.   /|   |
32.        |
33.        |
34. ========''', '''
35.
36.    +---+
37.    |   |
38.    O   |
39.   /|\  |
40.        |
41.        |
42. ========''', '''
43.
44.    +---+
45.    |   |
46.    O   |
47.   /|\  |
48.   /    |
49.        |
50. ========''', '''
51.
52.    +---+
53.    |   |
54.    O   |
55.   /|\  |
56.   / \  |
57.        |
58. ========''']
```

```
59. words = 'ant baboon badger bat bear beaver camel cat clam cobra cougar
coyote crow deer dog donkey duck eagle ferret fox frog goat goose hawk lion
lizard llama mole monkey moose mouse mule newt otter owl panda parrot pigeon
python rabbit ram rat raven rhino salmon seal shark sheep skunk sloth snake
spider stork swan tiger toad trout turkey turtle weasel whale wolf wombat
zebra'.split()
60.
61. def getRandomWord(wordList):
62.     # This function returns a random string from the passed list of
strings.
63.     wordIndex = random.randint(0, len(wordList) - 1)
64.     return wordList[wordIndex]
65.
66. def displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord):
67.     print(HANGMANPICS[len(missedLetters)])
68.     print()
69.
70.     print('Missed letters:', end=' ')
71.     for letter in missedLetters:
72.         print(letter, end=' ')
73.     print()
74.
75.     blanks = '_' * len(secretWord)
76.
77.     for i in range(len(secretWord)): # replace blanks with correctly
guessed letters
78.         if secretWord[i] in correctLetters:
79.             blanks = blanks[:i] + secretWord[i] + blanks[i+1:]
80.
81.     for letter in blanks: # show the secret word with spaces in between
each letter
82.         print(letter, end=' ')
83.     print()
84.
85. def getGuess(alreadyGuessed):
86.     # Returns the letter the player entered. This function makes sure the
player entered a single letter, and not something else.
87.     while True:
88.         print('Guess a letter.')
89.         guess = input()
90.         guess = guess.lower()
91.         if len(guess) != 1:
92.             print('Please enter a single letter.')
93.         elif guess in alreadyGuessed:
94.             print('You have already guessed that letter. Choose again.')
95.         elif guess not in 'abcdefghijklmnopqrstuvwxyz':
96.             print('Please enter a LETTER.')
```

```
 97.            else:
 98.                return guess
 99.
100. def playAgain():
101.     # This function returns True if the player wants to play again,
otherwise it returns False.
102.     print('Do you want to play again? (yes or no)')
103.     return input().lower().startswith('y')
104.
105.
106. print('H A N G M A N')
107. missedLetters = ''
108. correctLetters = ''
109. secretWord = getRandomWord(words)
110. gameIsDone = False
111.
112. while True:
113.     displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)
114.
115.     # Let the player type in a letter.
116.     guess = getGuess(missedLetters + correctLetters)
117.
118.     if guess in secretWord:
119.         correctLetters = correctLetters + guess
120.
121.         # Check if the player has won
122.         foundAllLetters = True
123.         for i in range(len(secretWord)):
124.             if secretWord[i] not in correctLetters:
125.                 foundAllLetters = False
126.                 break
127.         if foundAllLetters:
128.             print('Yes! The secret word is "' + secretWord + '"! You have
won!')
129.             gameIsDone = True
130.     else:
131.         missedLetters = missedLetters + guess
132.
133.         # Check if player has guessed too many times and lost
134.         if len(missedLetters) == len(HANGMANPICS) - 1:
135.             displayBoard(HANGMANPICS, missedLetters, correctLetters,
secretWord)
136.             print('You have run out of guesses!\nAfter ' +
str(len(missedLetters)) + ' missed guesses and ' + str(len(correctLetters)) + '
correct guesses, the word was "' + secretWord + '"')
137.             gameIsDone = True
138.
```

```
139.     # Ask the player if they want to play again (but only if the game is
done).
140.     if gameIsDone:
141.         if playAgain():
142.             missedLetters = ''
143.             correctLetters = ''
144.             gameIsDone = False
145.             secretWord = getRandomWord(words)
146.         else:
147.             break
```

How the Code Works

```
1. import random
```

The Hangman program randomly selected a secret word from a list of secret words. The `random` module will provide this ability, so line 1 imports it.

```
2. HANGMANPICS = ['''
3.
4.   +---+
5.   |   |
6.       |
7.       |
8.       |
9.       |
10. =========''', '''

...the rest of the code is too big to show here...
```

This one assignment statement stretches over lines 2 to 58 in the source code. To help you understand what this code means, let's learn about multi-line strings.

Multi-line Strings

So far all strings have been on one line and had one quote character at the start and end. However, if you use three quotes at the start and end then the string can go across several lines:

```
>>> fizz = '''Dear Alice,
I will return to Carol's house at the end of the month. I will see you then.
Your friend,
Bob'''
>>> print(fizz)
```

```
Dear Alice,
I will return to Carol's house at the end of the month. I will see you then.
Your friend,
Bob
```

These are **multi-line strings**. In a multi-line string, the newline characters are included as part of the string. You don't have to use the \n escape character, or escape quotes as long as you don't use three of them together. This makes the code easier to read for large amounts of text.

Constant Variables

The HANGMANPICS variable's name is in all capitals. This is the programming convention for constant variables. **Constants** are variables meant to have values that never changes from their first assignment statement. Although you can change the value in HANGMANPICS just like any other variable, the all-caps name reminds you to not do so. Since the HANGMANPICS variable never needs to change, it's marked as a constant.

Like all conventions, you don't *have* to follow it. But following this convention makes it easier for other programmers to read your code. They'll know that HANGMANPICS will always have the value it was assigned on line 2.

Lists

A **list** value can contain several other values inside it. Try entering this into the interactive shell:.

```
>>> spam = ['Life', 'The Universe', 'Everything', 42]
>>> spam
['Life', 'The Universe', 'Everything', 42]
```

This list value in spam contains four values. When typing the list value into your code, it begins with a [square bracket and ends with a] square bracket. This is like how strings begin and end with a quote character.

Commas separate the individual values inside of a list. These values are also called **items**.

Indexes

Try entering animals = ['aardvark', 'anteater', 'antelope', 'albert'] into the interactive shell to store a list in the variable animals. The square brackets are also used to access

an item inside a list. Try entering `animals[0]`, `animals[1]`, `animals[2]`, and `animals[3]` into the interactive shell to see how they evaluate:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[0]
'aardvark'
>>> animals[1]
'anteater'
>>> animals[2]
'antelope'
>>> animals[3]
'albert'
```

The number between the square brackets is the **index**. In Python, the index of the first item in a list is 0. The second item is at index 1, the third item is at index 2, and so on. Because the indexes begin at 0, not 1, we say that Python lists are **zero-indexed**.

Lists are good for storing several values without using a variable for each one. Otherwise, the code would look like this:

```
>>> animals1 = 'aardvark'
>>> animals2 = 'anteater'
>>> animals3 = 'antelope'
>>> animals4 = 'albert'
```

This code would be hard to manage if you have hundreds or thousands of strings. But a list can easily contain any number of values. Using the square brackets, you can treat items in the list just like any other value. Try entering `animals[0] + animals[2]` into the interactive shell:

```
>>> animals[0] + animals[2]
'aardvarkantelope'
```

The evaluation looks like this:

```
animals[0] + animals[2]
           ▼
'aardvark' + animals[2]
           ▼
'aardvark' + 'antelope'
           ▼
  'aardvarkantelope'
```

IndexError

If you try accessing an index that is too large, you'll get an **IndexError** that will crash your program. Try entering the following into the interactive shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[9999]
Traceback (most recent call last):
File "", line 1, in
animals[9999]
IndexError: list index out of range
```

Changing the Values of List Items with Index Assignment

You can also use the square brackets to change the value of an item in a list. Try entering the following into the interactive shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[1] = 'ANTEATER'
>>> animals
['aardvark', 'ANTEATER', 'antelope', 'albert']
```

The new `'ANTEATER'` string overwrites the second item in the `animals` list. So `animals[1]` will evaluate to the list's second item in expressions, but you can also use it on the left side of an assignment statement to assign a value as the list's second item.

List Concatenation

You can join lists into one list with the + operator, just like you can join strings. Joining lists with the + operator is **list concatenation**. Try entering the following into the interactive shell:

```
>>> [1, 2, 3, 4] + ['apples', 'oranges'] + ['Alice', 'Bob']
[1, 2, 3, 4, 'apples', 'oranges', 'Alice', 'Bob']
```

`['apples'] + ['oranges']` will evaluate to `['apples', 'oranges']`. But `['apples'] + 'oranges'` will result in an error. You cannot add a list value and string value instead of two list values. If you want to add non-list values to a list, use the `append()` method (described later).

The `in` Operator

The `in` operator can tell you if a value is in a list or not. Expressions that use the `in` operator return a Boolean value: `True` if the value is in the list and `False` if it isn't. Try entering the following into the interactive shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> 'antelope' in animals
```

```
True
```

The expression `'antelope' in animals` returns `True` because the string `'antelope'` is one of the values in the `animals` list. It is located at index 2.

But if you type the expression `'ant' in animals`, this will return `False` because the string `'ant'` doesn't exist in the list.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> 'antelope' in animals
True
>>> 'ant' in animals
False
```

The `in` operator also works for strings. It checks if one string exists in another. Try entering the following into the interactive shell:

```
>>> 'hello' in 'Alice said hello to Bob.'
True
```

Deleting Items from Lists with *del* Statements

A `del` statement will delete an item at a certain index from a list. Try entering the following into the interactive shell:

```
>>> spam = [2, 4, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 6, 8, 10]
```

Notice that when you deleted the item at index 1, the item that used to be at index 2 became the new value at index 1. The item that used to be at index 3 moved to be the new value at index 2. Everything above the deleted item moved down one index.

You can type `del spam[1]` again and again to keep deleting items from the list:

```
>>> spam = [2, 4, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 8, 10]
>>> del spam[1]
>>> spam
```

```
[2, 10]
```

The `del` statement is a statement, not a function or an operator. It doesn't have parentheses or evaluate to a return value.

Lists of Lists

Lists can contain other values, including other lists. Let's say you have a list of groceries, a list of chores, and a list of your favorite pies. You can put all three lists into another list. Try entering the following into the interactive shell:

```
>>> groceries = ['eggs', 'milk', 'soup', 'apples', 'bread']
>>> chores = ['clean', 'mow the lawn', 'go grocery shopping']
>>> favoritePies = ['apple', 'frumbleberry']
>>> listOfLists = [groceries, chores, favoritePies]
>>> listOfLists
[['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean', 'mow the lawn', 'go
grocery shopping'], ['apple', 'frumbleberry']]
```

To get an item inside the list of lists, you would use two sets of square brackets like this: `listOfLists[1][2]` which would evaluate to the string `'go grocery shopping'`.

This is because `listOfLists[1][2]` evaluates to `['clean', 'mow the lawn', 'go grocery shopping'][2]`. That finally evaluates to `'go grocery shopping'`:

```
listOfLists[1][2]
         ▼
[['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean', 'mow the lawn', 'go
grocery shopping'], ['apple', 'frumbleberry']][1][2]
         ▼
['clean', 'mow the lawn', 'go grocery shopping'][2]
         ▼
'go grocery shopping'
```

Figure 9-1 is another example of a list of lists, along with some of the indexes that point to the items. The arrows point to indexes of the inner lists themselves. The image is also flipped on its side to make it easier to read.

Methods

Methods are functions attached to a value. For example, all string values have a `lower()` method, which returns a copy of the string value in lowercase. You can call it like `'Hello'.lower()`, which returns `'hello'`. You cannot call `lower()` by itself and you do not

pass a string argument to `lower()` (as in `lower('Hello')`). You must attach the method call to a specific string value using a period. The next section describes string methods further.
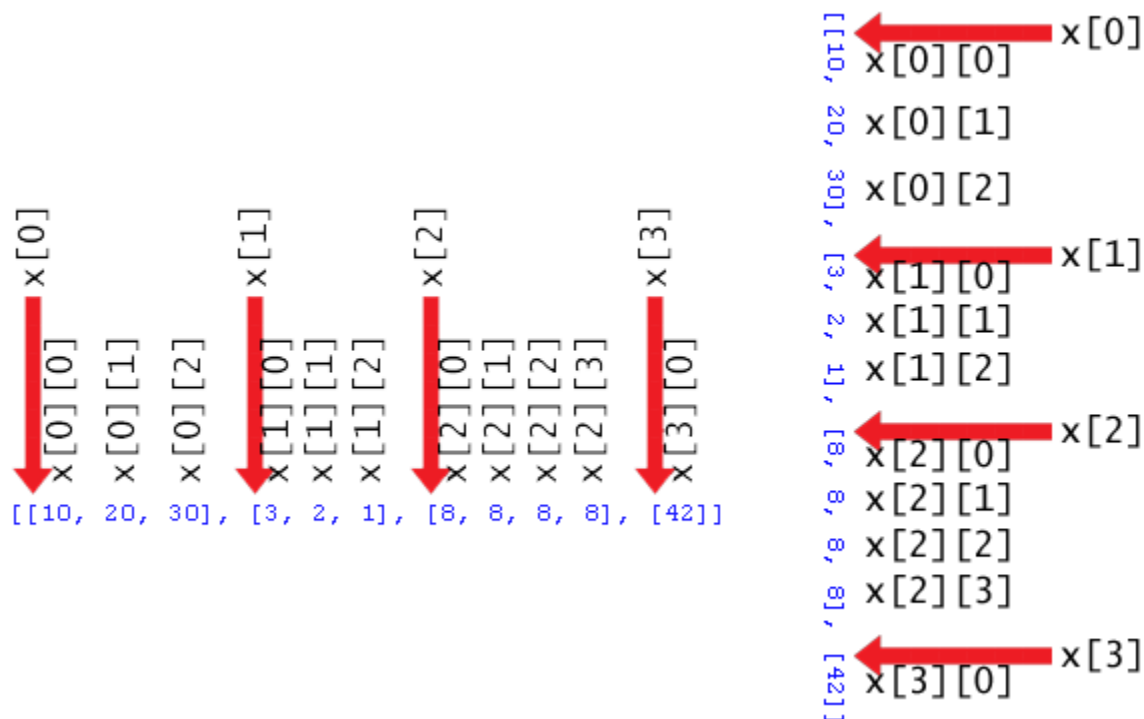
Figure 9-1: The indexes of a list of lists.

The `lower()` and `upper()` String Methods

Try entering `'Hello world!'.lower()` into the interactive shell to see an example of this method:

```
>>> 'Hello world!'.lower()
'hello world!'
```

There is also an `upper()` method for strings, which returns a string with all the characters in uppercase. Try entering `'Hello world!'.upper()` into the interactive shell:

```
>>> 'Hello world!'.upper()
'HELLO WORLD! '
```

Because the upper() method returns a string, you can call a method on that string also. Try entering 'Hello world!'.upper().lower() into the interactive shell:

```
>>> 'Hello world!'.upper().lower()
'hello world!'
```

'Hello world!'.upper() evaluates to the string 'HELLO WORLD!', and then string's lower() method is called. This returns the string 'hello world!', which is the final value in the evaluation.

```
'Hello world!'.upper().lower()
              ▼
      'HELLO WORLD!'.lower()
                ▼
            'hello world!'
```

The order is important. 'Hello world!'.lower().upper() isn't the same as 'Hello world!'.upper().lower():

```
>>> 'Hello world!'.lower().upper()
'HELLO WORLD!'
```

That evaluation looks like this:

```
'Hello world!'.lower().upper()
              ▼
      'hello world!'.upper()
                ▼
            'HELLO WORLD!'
```

If a string is stored in a variable, you can call a string method on that variable. Look at this example:

```
>>> spam = 'Hello world!'
>>> spam.upper()
'HELLO WORLD!'
```

This does not change the value in spam. The spam variable will still contain 'Hello world!'.

Note that the integer and float data types don't have any methods.

The `reverse()` and `append()` List Methods

The list data type also has methods. The `reverse()` method will reverse the order of the items in the list. Try entering `spam = [1, 2, 3, 4, 5, 6, 'meow', 'woof']`, and then `spam.reverse()` to reverse the list. Then enter `spam` to view the contents of the variable.

```
>>> spam = [1, 2, 3, 4, 5, 6, 'meow', 'woof']
>>> spam.reverse()
>>> spam
['woof', 'meow', 6, 5, 4, 3, 2, 1]
```

The most common list method you'll use is `append()`. This method will add the value you pass as an argument to the end of the list. Try entering the following into the interactive shell:

```
>>> eggs = []
>>> eggs.append('hovercraft')
>>> eggs
['hovercraft']
>>> eggs.append('eels')
>>> eggs
['hovercraft', 'eels']
>>> eggs.append(42)
>>> eggs
['hovercraft', 'eels', 42]
```

These methods do change the lists they are called on. They don't return a new list. We say that these methods change the list **in-place**.

The `split()` List Method

Line 59 is a long line of code, but it is really just a simple assignment statement. This line also uses the `split()` method, which is a method for the string data type like the `lower()` and `upper()` methods.

```
59. words = 'ant baboon badger bat bear beaver camel cat clam cobra cougar
coyote crow deer dog donkey duck eagle ferret fox frog goat goose hawk lion
lizard llama mole monkey moose mouse mule newt otter owl panda parrot pigeon
python rabbit ram rat raven rhino salmon seal shark sheep skunk sloth snake
spider stork swan tiger toad trout turkey turtle weasel whale wolf wombat
zebra'.split()
```

This assignment statement has just one long string, full of words separated by spaces. And at the end of the string is a split() method call. The split() method evaluates to a list with each word in the string as a single list item. The "split" occurs wherever a space occurs in the string.

It is easier to type the code using split(). If you created it as a list to begin with, you would have to type: ['ant', 'baboon', 'badger',... and so on, with quotes and commas for every word.

For example, try entering the following into the interactive shell:

```
>>> sentence = input()
My very energetic mother just served us nachos.
>>> sentence.split()
['My', 'very', 'energetic', 'mother', 'just', 'served', 'us', 'nachos.']
```

The result is a list of nine strings, one string for each of the words in the original string. The spaces are not included in any of the items in the list.

You can also add your own words to the string on line 59, or remove any you don't want to be in the game. Just make sure that spaces separate the words.

How the Code Works

Line 61 defines the getRandomWord() function. A list argument will be passed for its wordList parameter. This function will return a single secret word from the list in wordList.

```
61. def getRandomWord(wordList):
62.     # This function returns a random string from the passed list of
strings.
63.     wordIndex = random.randint(0, len(wordList) - 1)
64.     return wordList[wordIndex]
```

Line 63 stores a random index for this list in the wordIndex variable. You do this by calling randint() with two arguments. The first argument is 0 (for the first possible index) and the second argument is the value that the expression len(wordList) - 1 evaluates to (for the last possible index in a wordList).

List indexes start at 0, not 1. If you have a list of three items, the index of the first item is 0, the index of the second item is 1, and the index of the third item is 2. The length of this list is 3, but the index 3 would be after the last index. This is why line 63 subtracts 1 from the length. The code on line 63 will work no matter what the size of wordList is. Now you can add or remove strings to wordList if you like.

The `wordIndex` variable will be set to a random index for the list passed as the `wordlist` parameter. Line 64 will return the element in `wordList` at the integer index stored in `wordIndex`.

Let's pretend `['apple', 'orange', grape']` was passed as the argument to `getRandomWord()` and that `randint(0, 2)` returned the integer 2. That would mean that line 64 would evaluate to `return wordList[2]`, and then evaluate to `return 'grape'`. This is how the `getRandomWord()` returns a random string in the `wordList` list.

So the input to `getRandomWord()` is a list of strings, and the return value output is a randomly selected string from that list. This will be useful for the Hangman game to select a secret word for the player to guess.

Displaying the Board to the Player

Next, you need a function to print the hangman board on the screen. It will also display how many letters the player has correctly (and incorrectly) guessed.

```
66. def displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord):
67.     print(HANGMANPICS[len(missedLetters)])
68.     print()
```

This code defines a new function named `displayBoard()`. This function has four parameters:

- `HANGMANPICS` - A list of multi-line strings that will display the board as ASCII art. (The global `HANGMANPICS` variable will be passed for this parameter.)
- `missedLetters` - A string of the letters the player has guessed that are not in the secret word.
- `correctLetters` - A string of the letters the player has guessed that are in the secret word.
- `secretWord` – A string of the secret word that the player is trying to guess.

The first `print()` function call will display the board. `HANGMANPICS` will be a list of strings for each possible board. `HANGMANPICS[0]` shows an empty gallows, `HANGMANPICS[1]` shows the head (when the player misses one letter), `HANGMANPICS[2]` shows a head and body (when the player misses two letters), and so on until `HANGMANPICS[6]` which shows the full hangman.

The number of letters in `missedLetters` will reflect how many incorrect guesses the player has made. Call `len(missedLetters)` to find out this number. So, if `missedLetters` is `'aetr'` then `len('aetr')` will return 4. Printing `HANGMANPICS[4]` will display the appropriate hangman board for 4 misses. This is what `HANGMANPICS[len(missedLetters)]` on line 67 evaluates to.

```
70.      print('Missed letters:', end=' ')
71.      for letter in missedLetters:
72.          print(letter, end=' ')
73.      print()
```

Line 70 prints the string `'Missed letters:'` with a space character at the end instead of a newline. Remember that the keyword argument `end=' '` uses only one = sign (like `=`), not two (like `==`).

Line 71 is a new type of loop, called a `for` loop. A `for` loop often uses the `range()` function. Both are explained in the next two sections.

The `range()` and `list()` Functions

When called with one argument, `range()` will return a range object of integers from `0` up to (but not including) the argument. This range object can be converted to the more familiar list data type with the `list()` function. Try entering `list(range(10))` into the interactive shell:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list('Hello')
['H', 'e', 'l', 'l', 'o']
```

The `list()` function is similar to the `str()` or `int()` functions. It takes the value it is passed and returns a list. It's easy to generate huge lists with the `range()` function. Try entering in `list(range(10000))` into the interactive shell:

```
>>> list(range(10000))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,...
      ...skipped for brevity...
...9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
```

The list is so huge, that it won't even all fit onto the screen. But you can store the list into a variable:

```
>>> spam = list(range(10000))
```

If you pass two integer arguments to `range()`, the range object it returns is from the first integer argument up to (but not including) the second integer argument. Try entering `list(range(10, 20))` into the interactive shell:

```
>>> list(range(10, 20))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

The `range()` function is often used in `for` loops, which are much like the `while` loops you've already seen.

`for` Loops

The `for` loop is useful for looping over a list of values. This is different from the `while` loop, which loops as long as a certain condition is `True`. A `for` statement begins with the `for` keyword, followed by a new variable name, followed by the `in` keyword, followed by an iterable value, and ending with a colon.

An iterable is a value of the list, range, or string data types. There are also other data types that are considered iterables which will be introduced later.

Each time the program execution iterates through the loop the new variable in the `for` statement is assigned the value of the next item in the list.

```
>>> for i in range(5):
...     print('i is set to ' + str(i))
...
i is set to 0
i is set to 1
i is set to 2
i is set to 3
i is set to 4
```

The range object returned by `range(5)` is equivalent to the list `[0, 1, 2, 3, 4]` in a `for` statement. The first time the execution goes through the code in the `for`-block, the variable `i` will be set to `0`. On the next iteration, `i` will be set to `1`, and so on.

The `for` statement automatically converts the range object returned by `range()` into a list, so there's no need for `list(range(5))` in the `for` statement. Just use `range(5)`.

Lists and strings are also iterable data types. You can use them in `for` statements. Try entering the following into the interactive shell:

```
>>> for thing in ['cats', 'pasta', 'programming', 'spam']:
...     print('I really like ' + thing)
...
I really like cats
I really like pasta
I really like programming
I really like spam

>>> for i in 'Hello':
...     print(i)
```

```
...
H
e
l
l
o
```

A *while* Loop Equivalent of a *for* Loop

The for loop is similar to the while loop, but when you only need to iterate over items in a list, using a for loop is much less code to type. This is a while loop that acts the same as the previous for loop by adding extra code:

```
>>> iterableVal = ['cats', 'pasta', 'programming', 'spam']
>>> index = 0
>>> while (index < len(iterableVal)):
...     thing = iterableVal[index]
...     print('I really like ' + thing)
...     index = index + 1
...
I really like cats
I really like pasta
I really like programming
I really like spam
```

But using the for statement automatically does this extra code and makes programming easier since you have less to type.

The rest of the displayBoard() function displays the missed letters and creates the string of the secret word with all the not yet guessed letters as blanks.

```
70.     print('Missed letters:', end=' ')
71.     for letter in missedLetters:
72.         print(letter, end=' ')
73.     print()
```

The for loop on line 71 will iterate over each character in the missedLetters string and print them on the screen. Remember that the end=' ' will replace the newline character that is printed after the string with a single space character.

For example, if missedLetters was 'ajtw' this for loop would display a j t w.

Slicing

List slicing creates a new list value with a subset of another list's items. In code, specify two indexes (the beginning and end) with a colon in the square brackets after a list. For example, try entering the following into the interactive shell:

```
>>> spam = ['apples', 'bananas', 'carrots', 'dates']
>>> spam[1:3]
['bananas', 'carrots']
```

The expression spam[1:3] evaluates to a list with items from index 1 up to (but not including) index 3 in spam.

If you leave out the first index, Python will automatically think you want index 0 for the first index:

```
>>> spam = ['apples', 'bananas', 'carrots', 'dates']
>>> spam[:2]
['apples', 'bananas']
```

If you leave out the second index, Python will automatically think you want the rest of the list:

```
>>> spam = ['apples', 'bananas', 'carrots', 'dates']
>>> spam[2:]
['carrots', 'dates']
```

Slicing is a simple way to get a subset of the items in a list. You use slices with strings in the same way you use them with lists. Each character in the string is like an item in the list. Try entering the following into the interactive shell:

```
>>> myName = 'Zophie the Fat Cat'
>>> myName[4:12]
'ie the F'
>>> myName[:10]
'Zophie the'
>>> myName[7:]
'the Fat Cat'
```

The next part of the code in Hangman uses slicing.

Displaying the Secret Word with Blanks

Now you want code to print the secret word, but with blank lines for the letters that have not been guessed. You can use the _ character (called the underscore character) for this. First create a

string with nothing but one underscore for each letter in the secret word. Then replace the blanks for each letter in correctLetters.

So if the secret word was 'otter' then the blanked out string would be '_____' (five _ characters). If correctLetters was the string 'rt' you would change the string to '_tt_r'. Line 75 to 79 is the code that does that.

```
75.      blanks = '_' * len(secretWord)
```

Line 75 creates the blanks variable full of _ underscores using string replication. Remember that the * operator can also be used on a string and an integer, so the expression '_' * 5 evaluates to '_____'. This will make sure that blanks has the same number of underscores as secretWord has letters.

```
77.      for i in range(len(secretWord)): # replace blanks with correctly
guessed letters
78.          if secretWord[i] in correctLetters:
79.              blanks = blanks[:i] + secretWord[i] + blanks[i+1:]
```

Line 77 has a for loop to go through each letter in secretWord and replace the underscore with the actual letter if it exists in correctLetters.

For example, pretend the value of secretWord is 'otter' and the value in correctLetters is 'tr'. You would want the string '_tt_r' displayed to the player. Let's figure out how to create this string.

Line 77's len(secretWord) call would return 5. The range(len(secretWord)) call becomes range(5), which makes the for loop iterate over 0, 1, 2, 3, and 4.

Because the value of i will take on each value in [0, 1, 2, 3, 4], the code in the for loop is the same as this:

```
if secretWord[0] in correctLetters:
    blanks = blanks[:0] + secretWord[0] + blanks[1:]

if secretWord[1] in correctLetters:
    blanks = blanks[:1] + secretWord[1] + blanks[2:]

if secretWord[2] in correctLetters:
    blanks = blanks[:2] + secretWord[2] + blanks[3:]

if secretWord[3] in correctLetters:
    blanks = blanks[:3] + secretWord[3] + blanks[4:]
```

```
if secretWord[4] in correctLetters:
    blanks = blanks[:4] + secretWord[4] + blanks[5:]
```

If you are confused as to what the value of something like secretWord[0] or blanks[3:] is, then look at Figure 9-2. It shows the value of the secretWord and blanks variables, and the index for each letter in the string.
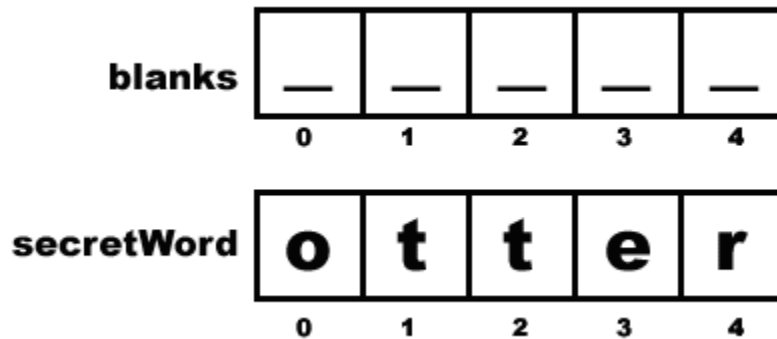
Figure 9-2: The indexes of the blanks and secretWord strings.

If you replace the list slices and the list indexes with the values that they represent, the loop code would be the same as this:

```
if 'o' in 'tr': # False
    blanks = '' + 'o' + '____' # This line is skipped.

if 't' in 'tr': # True
    blanks = '_' + 't' + '___' # This line is executed.

if 't' in 'tr': # True
    blanks = '_t' + 't' + '__' # This line is executed.

if 'e' in 'tr': # False
    blanks = '_tt' + 'e' + '_' # This line is skipped.

if 'r' in 'tr': # True
    blanks = '_tt_' + 'r' + '' # This line is executed.

# blanks now has the value '_tt_r'
```

The above code examples all do the *same thing* when secretWord is 'otter' and correctLetters is 'tr'. The next few lines of code print the new value of blanks with spaces between each letter.

```
81.     for letter in blanks: # show the secret word with spaces in between
each letter
82.         print(letter, end=' ')
83.     print()
```

Get the Player's Guess

The `getGuess()` function will be called so that the player can enter a letter to guess. The function returns the letter the player guessed as a string. Further, `getGuess()` will make sure that the player types a valid letter before returning from the function.

```
85. def getGuess(alreadyGuessed):
86.     # Returns the letter the player entered. This function makes sure the
player entered a single letter, and not something else.
```

A string of the letters the player has guessed is passed as the argument for the `alreadyGuessed` parameter. Then the `getGuess()` function asks the player to guess a single letter. This single letter will be `getGuess()`'s return value.

```
87.     while True:
88.         print('Guess a letter.')
89.         guess = input()
90.         guess = guess.lower()
```

Line 87's `while` loop will keep asking the player for a letter until they enter text that is:

1. A single letter.

2. A letter they have not guessed previously.

The condition for the `while` loop is simply the Boolean value `True`. That means the only way execution will ever leave this loop is by executing a `break` statement (which leaves the loop) or a `return` statement (which leaves not just the loop but the entire function).

The code inside the loop asks the player to enter a letter, which is stored in the variable `guess`. If the player entered a capitalized letter, it will be overwritten with a to lowercase letter on line 90.

`elif` ("Else If") Statements

The next part of the Hangman program uses `elif` statements. You can think of `elif` "else if" statements as saying "If this is true, do this. Or else if this next condition is true, do that. Or else if none of them are true, do this last thing."

Take a look at the following code:

```
if catName == 'Fuzzball':
    print('Your cat is fuzzy.')
elif catName == 'Spots':
        print('Your cat is spotted.')
else:
    print('Your cat is not fuzzy or spotted.')
```

If the `catName` variable is equal to the string `'Fuzzball'`, then the `if` statement's condition is `True` and the if-block tells the user that their cat is fuzzy. However, if this condition is `False`, then Python tries the `elif` ("else if") statement's condition next. If `catName` is `'Spots'`, then the `'Your cat is spotted.'` string is printed to the screen. If both are `False`, then the code tells the user their cat isn't fuzzy or spotted.

You can have as many `elif` statements as you want:

```
if catName == 'Fuzzball':
    print('Your cat is fuzzy.')
elif catName == 'Spots':
    print('Your cat is spotted.')
elif catName == 'Chubs':
    print('Your cat is chubby.')
elif catName == 'Puff':
    print('Your cat is puffy.')
else:
    print('Your cat is neither fuzzy nor spotted nor chubby nor puffy.')
```

When one of the `elif` conditions is `True`, its code is executed and then execution jumps to the first line past the else-block. So *one and only one* of the blocks in the `if-elif-else` statements will be executed. You can also leave off the `else`-block if you don't need one, and just have `if-elif` statements.

Making Sure the Player Entered a Valid Guess

```
91.         if len(guess) != 1:
92.             print('Please enter a single letter.')
93.         elif guess in alreadyGuessed:
94.             print('You have already guessed that letter. Choose again.')
95.         elif guess not in 'abcdefghijklmnopqrstuvwxyz':
96.             print('Please enter a LETTER.')
97.         else:
98.             return guess
```

The guess variable contains player's letter guess. The program needs to make sure they typed in a valid guess: one and only one lowercase letter. If they didn't, the execution should loop back and ask them for a letter again.

Line 91's condition checks if guess is not one character long. Line 93's condition checks if guess already exists inside the alreadyGuessed variable. Line 95's condition checks if guess is not a lowercase letter.

If all of these conditions are False, then the else statement's block executes and getGuess() returns the value in guess on line 98.

Remember, only one of the blocks in if-elif-else statements will be executed.

Asking the Player to Play Again

```
100. def playAgain():
101.     # This function returns True if the player wants to play again,
otherwise it returns False.
102.     print('Do you want to play again? (yes or no)')
103.     return input().lower().startswith('y')
```

The playAgain() function has just a print() function call and a return statement. The return statement has an expression that looks complicated, but you can break it down. Here's a step by step look at how Python evaluates this expression if the user types in YES.

```
input().lower().startswith('y')
              ▼
  'YES'.lower().startswith('y')
              ▼
         'yes'.startswith('y')
              ▼
            True
```

The point of the playAgain() function is to let the player type in yes or no to tell the program if they want to play another round of Hangman. The player should be able to type YES, yes, Y, or anything else that begins with a "Y" in order to mean "yes". If the player types in YES, then the return value of input() is the string 'YES'. And 'YES'.lower() returns the lowercase version of the attached string. So the return value of 'YES'.lower() is 'yes'.

But there's the second method call, startswith('y'). This function returns True if the associated string begins with the string parameter between the parentheses, and False if it doesn't. The return value of 'yes'.startswith('y') is True.

Now you have evaluated this expression! What it does is let the player type in a response, lowercases the response, checks if it begins with the letter `'y'`, then returns `True` if it does and `False` if it doesn't.

On a side note, there's also an `endswith(someString)` string method that will return `True` if the string ends with the string in `someString` and `False` if it doesn't. `endswith()` is sort of like the opposite of `startswith()`.

Review of the Hangman Functions

That's all the functions we are creating for this game! Let's review them:

- `getRandomWord(wordList)` will take a list of strings passed to it, and return one string from it. That is how a word is chosen for the player to guess.
- `displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)` will show the current state of the board, including how much of the secret word the player has guessed so far and the wrong letters the player has guessed. This function needs four parameters passed to work correctly. `HANGMANPICS` is a list of strings that hold the ASCII art for each possible hangman board. `correctLetters` and `missedLetters` are strings made up of the letters that the player has guessed that are in and not in the secret word, respectively. And `secretWord` is the secret word the player is trying to guess. This function has no return value.
- `getGuess(alreadyGuessed)` takes a string of letters the player has already guessed and will keep asking the player for a letter that isn't in `alreadyGuessed`.) This function returns the string of the valid letter the player guessed.
- `playAgain()` is a function that asks if the player wants to play another round of Hangman. This function returns `True` if the player does and `False` if the player doesn't.

After the functions is the code for the main part of the program at line 106. Everything previous was just function definitions and a large assignment statement for `HANGMANPICS`.

Setting Up the Variables

```
106. print('H A N G M A N')
107. missedLetters = ''
108. correctLetters = ''
109. secretWord = getRandomWord(words)
110. gameIsDone = False
```

Line 106 is the first `print()` call that executes when the game is run. It displays the title of the game. Next is assigning blank strings for `missedLetters` and `correctLetters` since the player hasn't guessed any missed or correct letters yet.

The `getRandomWord(words)` call will evaluate to a randomly selects word from the `words` list.

Line 110 sets `gameIsDone` to `False`. The code will set `gameIsDone` to `True` when it wants to signal that the game is over and should ask the player if they want to play again.

Displaying the Board to the Player

```
112. while True:
113.     displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)
```

The `while` loop's condition is always `True`, which means it will loop forever until a `break` statement is encountered. (This happens later on line 147.)

Line 113 calls the `displayBoard()` function, passing it the list of hangman ASCII art pictures and the three variables set on lines 107, 108, and 109. Based on how many letters the player has correctly guessed and missed, this function displays the appropriate hangman board to the player.

Letting the Player Enter Their Guess

```
115.     # Let the player type in a letter.
116.     guess = getGuess(missedLetters + correctLetters)
```

The `getGuess()` function needs all the letters in `missedLetters` and `correctLetters` combined, so line 116 concatenates the strings in these variables and passes the result as the argument. This argument is needed by `getGuess()` because the function has to check if the player types in a letter that they have already guessed.

Checking if the Letter is in the Secret Word

```
118.     if guess in secretWord:
119.         correctLetters = correctLetters + guess
```

If the `guess` string exists in `secretWord`, then concatenate `guess` to the end of the `correctLetters` string. This string will be the new value of `correctLetters`.

Checking if the Player has Won

```
121.         # Check if the player has won
122.         foundAllLetters = True
123.         for i in range(len(secretWord)):
124.             if secretWord[i] not in correctLetters:
125.                 foundAllLetters = False
126.                 break
```

How can the program know if the player has guessed every letter in the secret word? Well, correctLetters has each letter that the player correctly guessed and secretWord is the secret word itself. But you can't just check if correctLetters == secretWord because consider this case: if secretWord was the string 'otter' and correctLetters was the string 'orte', then correctLetters == secretWord would be False even though the player *has* guessed each letter in the secret word.

The only way you can be sure the player won is to iterate over each letter in secretWord and see if it exists in correctLetters. If, and only if, every letter in secretWord exists in correctLetters will the player have won.

If you find a letter in secretWord that doesn't exist in correctLetters, you know that the player has **not** guessed all the letters. The new variable foundAllLetters is set to True on line 122 before the loop begins. The loop starts out assuming that all the letters in the secret word have been found. But the loop's code on line 125 will change foundAllLetters to False the first time it finds a letter in secretWord that isn't in correctLetters.

```
127.         if foundAllLetters:
128.             print('Yes! The secret word is "' + secretWord + '"! You have
won!')
129.             gameIsDone = True
```

If all letters in the secret word have been found, the player is told they have won and gameIsDone is set to True.

When the Player Guesses Incorrectly

```
130.     else:
131.         missedLetters = missedLetters + guess
```

This is the start of the else-block. Remember, the code in this block will execute if the condition was False. But which condition? To find out, point your finger at the start of the else keyword

and move it straight up like in Figure 9-3. You'll see that the `else` keyword's indentation is the same as the `if` keyword's indentation on line 118.

Figure 9-3: The `else` statement is matched with the `if` statement at the same indentation.

So if the condition on line 118 (`guess in secretWord`) was `False`, then the execution moves into this else-block.

Wrongly guessed letters are concatenated to the `missedLetters` string on line 131. This is like what line 119 did for letters the player guessed correctly.

```
133.           # Check if player has guessed too many times and lost
134.           if len(missedLetters) == len(HANGMANPICS) - 1:
135.               displayBoard(HANGMANPICS, missedLetters, correctLetters,
secretWord)
136.               print('You have run out of guesses!\nAfter ' +
str(len(missedLetters)) + ' missed guesses and ' + str(len(correctLetters)) + '
correct guesses, the word was "' + secretWord + '"')
137.               gameIsDone = True
```

Each time the player guesses wrong, the code concatenates the wrong letter to the string in `missedLetters`. So the length of `missedLetters` (or, in code, `len(missedLetters)`) is also the number of wrong guesses.

The `HANGMANPICS` list has 7 ASCII art strings. So when `len(missedLetters)` equals 6, you know the player has lost because the hangman picture will be finished. Remember, `HANGMANPICS[0]` is the first item in the list, and `HANGMANPICS[6]` is the last one.

So, when the length of the `missedLetters` string is equal to `len(HANGMANPICS) - 1` (that is, 6), the player has run out of guesses. Line 136 prints the secret word and line 137 sets the `gameIsDone` variable is set to `True`.

```
139.     # Ask the player if they want to play again (but only if the game is
done).
140.    if gameIsDone:
141.        if playAgain():
142.            missedLetters = ''
143.            correctLetters = ''
144.            gameIsDone = False
145.            secretWord = getRandomWord(words)
```

If the player won or lost after guessing their letter, the game should ask the player if they want to play again. The playAgain() function handles getting a yes or no from the player, so it is called on line 141.

If the player does want to play again, the values in missedLetters and correctLetters must be reset to blank strings, gameIsDone to False, and a new secret word stored in secretWord. This way when the execution loops back to the beginning of the while loop on line 112, the board will be back to a fresh game.

```
146.        else:
147.            break
```

If the player did not type in something that began with "y" when asked if they wanted to play again, then line 141's condition would be False and the else-block executes. The break statement causes the execution to jump to the first instruction after the loop. But because there are no more instructions after the loop, the program terminates.

Summary

This has been a long chapter, and you've been introduced to several new concepts. But Hangman has been our most advanced game yet. As your games get more and more complex, it'll be a good idea to sketch out a flow chart on paper of what happens in your program.

Lists are values that can contain other values. Methods are functions specific to a data type. Lists have append() and reverse() methods. Strings have lower(), upper(), split(), startswith(), and endswith() methods. You'll learn about many more data types and methods in the rest of this book.

The for loop is a loop that iterates over the items in a list, unlike a while loop which iterates as long as a condition is True. The elif statement lets you add an "or else if" clause to the middle of your if-else statements. The del statement can delete variables or items inside lists.

Chapter 9 ½

EXTENDING HANGMAN

Topics Covered In This Chapter:
- The dictionary data type
- Key-value pairs
- The keys() and values() dictionary methods
- Multiple variable assignment

The Hangman is much bigger than the Dragon Realm program, but it's is also more sophisticated. It really helps to make a flow chart or small sketch to remember how you want everything to work. Now that you've created a basic Hangman game, let's look at some ways you can extend it with new features.

After you've played Hangman a few times, you might think that six guesses aren't enough to get many of the words. You can easily give the player more guesses by adding more multi-line strings to the HANGMANPICS list.

Save your *hangman.py* program as *hangman2.py*, then add the following instructions:

```
58. =========''', '''
59.  +----+
60.  |    |
61. [O    |
62. /|\   |
63. / \   |
64.       |
65. =========''', '''
66.  +----+
67.  |    |
68. [O]   |
69. /|\   |
70. / \   |
71.       |
72. =========''']
```

There are two new multi-line strings to the HANGMANPICS list, one with the hangman's left ear drawn, and the other with both ears drawn. Because the program will tell the player they have lost on line134 based on len(missedLetters) == len(HANGMANPICS) - 1, this is the only change you must make. The rest of the program works with the new HANGMANPICS list just fine.

You can also change the list of words by changing the words on line 59. Instead of animals, you could have colors:

```
59. words = 'red orange yellow green blue indigo violet white black
brown'.split()
```

Or shapes:

```
59. words = 'square triangle rectangle circle ellipse rhombus trapezoid chevron
pentagon hexagon septagon octagon'.split()
```

Or fruits:

```
59. words = 'apple orange lemon lime pear watermelon grape grapefruit cherry
banana cantaloupe mango strawberry tomato'.split()
```

Dictionaries

With some modification, you can change the code so that the Hangman game uses sets of words, such as animal, color, shape, or fruit. The program can tell the player which set (animal, color, shape, or fruit) the secret word is from.

To make this change, you will need a new data type called a **dictionary**. A dictionary is a collection of values like a list is. But instead of accessing the items in the dictionary with an integer index, you can access them with an index of any data type. For dictionaries, these indexes are called **keys**.

Dictionaries use { and } curly braces instead of [and] square brackets. Try entering the following into the interactive shell:

```
>>> spam = {'hello':'Hello there, how are you?', 4:'bacon', 'eggs':9999 }
```

The values between the curly braces are **key-value pairs**. The keys are on the left of the colon and the key's values are on the right. You can access the values like items in lists by using the key. Try entering the following into the interactive shell:

```
>>> spam = {'hello':'Hello there, how are you?', 4:'bacon', 'eggs':9999}
>>> spam['hello']
'Hello there, how are you?'
>>> spam[4]
'bacon'
>>> spam[eggs]
9999
```

Instead of putting an integer between the square brackets, you can use, say, a string key. This will evaluate to the value for that key.

Getting the Size of Dictionaries with `len()`

You can get the number of key-value pairs in the dictionary with the `len()` function. Try entering the following into the interactive shell:

```
>>> stuff = {'hello':'Hello there, how are you?', 4:'bacon', 'spam':9999}
>>> len(stuff)
3
```

The Difference Between Dictionaries and Lists

Dictionaries can have keys of any data type, not just strings. But remember, because 0 and '0' are different values, they will be different keys. Try entering this into the interactive shell:

```
>>> spam = {'0':'a string', 0:'an integer'}
>>> spam[0]
'an integer'
>>> spam['0']
'a string'
```

The keys in dictionaries can also be looped over using a `for` loop. Try entering the following into the interactive shell.

```
>>> favorites = {'fruit':'apples', 'animal':'cats', 'number':42}
>>> for k in favorites:
...     print(k)
fruit
number
animal
>>> for k in favorites:
...     print(favorites[k])
apples
42
cats
```

Dictionaries are different from lists because the values inside them are unordered. The first item in a list named `listStuff` would be `listStuff[0]`. But there's no "first" item in a dictionary, because dictionaries do not have any sort of order. Try entering the following into the interactive shell:

```
>>> favorites1 = {'fruit':'apples', 'number':42, 'animal':'cats'}
```

```
>>> favorites2 = {'animal':'cats',  'number':42, 'fruit':'apples'}
>>> favorites1 == favorites2
True
```

The expression `favorites1 == favorites2` evaluates to `True` because dictionaries are unordered and considered equal if they have the same key-value pairs in them. Meanwhile, lists are ordered, so two lists with the same values in a different order are not equal to each other. Try entering this into the interactive shell:

```
>>> listFavs1 = ['apples', 'cats', 42]
>>> listFavs2 = ['cats', 42, 'apples']
>>> listFavs1 == listFavs2
False
```

Dictionaries have two useful methods, `keys()` and `values()`. These will return values of a type called `dict_keys` and `dict_values`, respectively. Much like range objects, values of those data types are returned by the `list()` function. Try entering the following into the interactive shell:

```
>>> favorites = {'fruit':'apples', 'animal':'cats', 'number':42}
>>> list(favorites.keys())
['fruit', 'number', 'animal']
>>> list(favorites.values())
['apples', 42, 'cats']
```

Sets of Words for Hangman

Let's change the code in the Hangman game to support different sets of secret words. First, replace the value assigned to `words` with a dictionary whose keys are strings and values are lists of strings. The string method `split()` will return a list of strings with one word each.

```
59. words = {'Colors':'red orange yellow green blue indigo violet white black
brown'.split(),
60. 'Shapes':'square triangle rectangle circle ellipse rhombus trapezoid
chevron pentagon hexagon septagon octagon'.split(),
61. 'Fruits':'apple orange lemon lime pear watermelon grape grapefruit cherry
banana cantaloupe mango strawberry tomato'.split(),
62. 'Animals':'bat bear beaver cat cougar crab deer dog donkey duck eagle fish
frog goat leech lion lizard monkey moose mouse otter owl panda python rabbit
rat shark sheep skunk squid tiger turkey turtle weasel whale wolf wombat
zebra'.split()}
```

Lines 59 to 62 are across multiple lines in the source code, but they are still one assignment statement. The instruction doesn't end until the final } curly brace on line 62.

The `random.choice()` Function

The `choice()` function in the `random` module takes a list argument and returns a random value from it. This is similar to the what the previous `getRandomWord()` function did. You'll use `random.choice()` in the new version of the `getRandomWord()` function.

To see how the `random.choice()` function works, try entering the following into the interactive shell:

```
>>> import random
>>> random.choice(['cat', 'dog', 'mouse'])
'mouse'
>>> random.choice(['cat', 'dog', 'mouse'])
'cat'
>>> random.choice([2, 22, 222, 223])
2
>>> random.choice([2, 22, 222, 223])
222
```

Change the `getRandomWord()` function so that its parameter will be a dictionary of lists of strings, instead of just a list of strings. Here is what the function originally looked like:

```
61. def getRandomWord(wordList):
62.     # This function returns a random string from the passed list of
strings.
63.     wordIndex = random.randint(0, len(wordList) - 1)
64.     return wordList[wordIndex]
```

Change the code in this function so that it looks like this:

```
64. def getRandomWord(wordDict):
65.     # This function returns a random string from the passed dictionary of
lists of strings, and the key also.
66.     # First, randomly select a key from the dictionary:
67.     wordKey = random.choice(list(wordDict.keys()))
68.
69.     # Second, randomly select a word from the key's list in the dictionary:
70.     wordIndex = random.randint(0, len(wordDict[wordKey]) - 1)
71.
72.     return [wordDict[wordKey][wordIndex], wordKey]
```

The name of the `wordList` parameter is changed to `wordDict` to be more descriptive. Now instead of choosing a random word from a list of strings, first the function chooses a random key in the `wordDict` dictionary by calling `random.choice()`.

And instead of returning the string `wordList[wordIndex]`, the function returns a list with two items. The first item is `wordDict[wordKey][wordIndex]`. The second item is `wordKey`.

Evaluating a Dictionary of Lists

The `wordDict[wordKey][wordIndex]` expression on line 72 may look complicated, but it is just an expression you can evaluate one step at a time like anything else. First, imagine that `wordKey` had the value `'Fruits'` (which was chosen on line 65) and `wordIndex` has the value 5 (chosen on line 68). Here is how `wordDict[wordKey][wordIndex]` would evaluate:

```
wordDict[wordKey][wordIndex]
            ▼
wordDict['Fruits'][wordIndex]
            ▼
['apple', 'orange', 'lemon', 'lime', 'pear', 'watermelon', 'grape',
'grapefruit', 'cherry', 'banana', 'cantaloupe', 'mango', 'strawberry',
'tomato'][wordIndex]
            ▼
['apple', 'orange', 'lemon', 'lime', 'pear', 'watermelon', 'grape',
'grapefruit', 'cherry', 'banana', 'cantaloupe', 'mango', 'strawberry',
'tomato'][5]
            ▼
     'watermelon'
```

In the above case, the item in the list this function returns would be the string `'watermelon'`. (Remember that indexes start at 0, so [5] refers to the 6th item in the list, not the 5th.)

Because the `getRandomWord()` function now returns a list of two items instead of a string, `secretWord` will be assigned a list, not a string. You can assign these two items into two separate variables using multiple assignment. This is explained next.

Multiple Assignment

Multiple assignment is a shortcut to specify multiple variables, separated by commas, on the left side of an assignment statement. Try entering the following into the interactive shell:

```
>>> a, b, c = ['apples', 'cats', 42]
>>> a
'apples'
>>> b
'cats'
>>> c
42
```

The above example is equivalent to the following assignment statements:

```
>>> a = ['apples', 'cats', 42][0]
>>> b = ['apples', 'cats', 42][1]
>>> c = ['apples', 'cats', 42][2]
```

You must put the same number of variables on the left side of the = assignment operator as there are items in the list on the right side. Python will automatically assign the first item's value in the list to the first variable, the second item's value to the second variable, and so on. But if you do not have the same number of variables and items, the Python interpreter will give you an error.

```
>>> a, b, c, d = ['apples', 'cats', 42, 10, 'hello']
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    a, b, c, d = ['apples', 'cats', 42, 10, 'hello']
ValueError: too many values to unpack

>>> a, b, c, d = ['apples', 'cats']
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    a, b, c = ['apples', 'cats']
ValueError: need more than 2 values to unpack
```

Change your code in Hangman on line 109 and 145 to use multiple assignment with the return value of getRandomWord():

```
108. correctLetters = ''
109. secretWord, secretKey = getRandomWord(words)
110. gameIsDone = False
...
144. gameIsDone = False
145. secretWord, secretKey = getRandomWord(words)
146. else:
```

Printing the Word Category for the Player

The last change you'll make is to tell the player which set of words they are trying to guess. This way, when the player plays the game they will know if the secret word is an animal, color, shape, or fruit. Add this line of code after line 112. Here is the original code:

```
112. while True:
113.     displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)
```

Add the line so your program looks like this:

```
112. while True:
113.     print('The secret word is in the set: ' + secretKey)
114.     displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)
```

Now you're done with the changes to the Hangman program. Instead of just a single list of strings, the secret word is chosen from many different lists of strings. The program also tells the player which set of words the secret word is from. Try playing this new version. You can easily change the words dictionary on line 59 to include more sets of words.

Summary

We're done with Hangman. Even after you've finished writing a game, you can always add more features after you learn more about Python programming.

Dictionaries are similar to lists except that they can use any type of value for an index, not just integers. The indexes in dictionaries are called keys.

Multiple assignment is a shortcut to assign multiple variables the values in a list.

Hangman was fairly advanced compared to the previous games in this book. But at this point, you know most of the basic concepts in writing programs: variables, loops, functions, and Python's data types such as lists and dictionaries. The later programs in this book will still be a challenge to master, but you have finished the steepest part of the climb.

Chapter 10

TIC TAC TOE

This chapter features a Tic Tac Toe game against a simple artificial intelligence. An **artificial intelligence** (or **AI**) is a computer program that can intelligently respond to the player's moves. This game doesn't introduce any complicated new concepts. The artificial intelligence that plays Tic Tac Toe is really just a few lines of code.

Two people play Tic Tac Toe with paper and pencil. One player is X and the other player is O. Players take turns placing their X or O. If a player gets three of their marks on the board in a row, column or one of the two diagonals, they win. When the board fills up with neither player winning, the game ends in a draw.

This chapter doesn't introduce many new programming concepts. It makes use of our existing programming knowledge to make an intelligent Tic Tac Toe player. Let's get started by looking at a sample run of the program. The player makes their move by entering the number of the space they want to go. These numbers are in the same places as the number keys on your keyboard's keypad (see Figure 10-2).

Sample Run of Tic Tac Toe

```
Welcome to Tic Tac Toe!
Do you want to be X or O?
X
The computer will go first.
   |   |
 O |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
```

```
    |   |
    |   |
    |   |
What is your next move? (1-9)
3
    |   |
 O |   |
    |   |
-----------
    |   |
    |   |
    |   |
-----------
    |   |
 O |   | X
    |   |
What is your next move? (1-9)
4
    |   |
 O |   | O
    |   |
-----------
    |   |
 X |   |
    |   |
-----------
    |   |
 O |   | X
    |   |
What is your next move? (1-9)
5
    |   |
 O | O | O
    |   |
-----------
    |   |
 X | X |
    |   |
-----------
    |   |
 O |   | X
    |   |
The computer has beaten you! You lose.
Do you want to play again? (yes or no)
no
```

Source Code of Tic Tac Toe

In a new file editor window, type in the following source code and save it as *tictactoe.py*. Then run the game by pressing **F5**.

```
                                                               tictactoe.py
 1. # Tic Tac Toe
 2.
 3. import random
 4.
 5. def drawBoard(board):
 6.     # This function prints out the board that it was passed.
 7.
 8.     # "board" is a list of 10 strings representing the board (ignore index
0)
 9.     print('   |   |')
10.     print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
11.     print('   |   |')
12.     print('-----------')
13.     print('   |   |')
14.     print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
15.     print('   |   |')
16.     print('-----------')
17.     print('   |   |')
18.     print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
19.     print('   |   |')
20.
21. def inputPlayerLetter():
22.     # Lets the player type which letter they want to be.
23.     # Returns a list with the player's letter as the first item, and the
computer's letter as the second.
24.     letter = ''
25.     while not (letter == 'X' or letter == 'O'):
26.         print('Do you want to be X or O?')
27.         letter = input().upper()
28.
29.     # the first element in the list is the player's letter, the second is
the computer's letter.
30.     if letter == 'X':
31.         return ['X', 'O']
32.     else:
33.         return ['O', 'X']
34.
35. def whoGoesFirst():
36.     # Randomly choose the player who goes first.
37.     if random.randint(0, 1) == 0:
```

```
38.         return 'computer'
39.     else:
40.         return 'player'
41.
42. def playAgain():
43.     # This function returns True if the player wants to play again,
otherwise it returns False.
44.     print('Do you want to play again? (yes or no)')
45.     return input().lower().startswith('y')
46.
47. def makeMove(board, letter, move):
48.     board[move] = letter
49.
50. def isWinner(bo, le):
51.     # Given a board and a player's letter, this function returns True if
that player has won.
52.     # We use bo instead of board and le instead of letter so we don't have
to type as much.
53.     return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the
top
54.     (bo[4] == le and bo[5] == le and bo[6] == le) or # across the middle
55.     (bo[1] == le and bo[2] == le and bo[3] == le) or # across the bottom
56.     (bo[7] == le and bo[4] == le and bo[1] == le) or # down the left side
57.     (bo[8] == le and bo[5] == le and bo[2] == le) or # down the middle
58.     (bo[9] == le and bo[6] == le and bo[3] == le) or # down the right side
59.     (bo[7] == le and bo[5] == le and bo[3] == le) or # diagonal
60.     (bo[9] == le and bo[5] == le and bo[1] == le)) # diagonal
61.
62. def getBoardCopy(board):
63.     # Make a duplicate of the board list and return it the duplicate.
64.     dupeBoard = []
65.
66.     for i in board:
67.         dupeBoard.append(i)
68.
69.     return dupeBoard
70.
71. def isSpaceFree(board, move):
72.     # Return true if the passed move is free on the passed board.
73.     return board[move] == ' '
74.
75. def getPlayerMove(board):
76.     # Let the player type in their move.
77.     move = ' '
78.     while move not in '1 2 3 4 5 6 7 8 9'.split() or not
isSpaceFree(board, int(move)):
79.         print('What is your next move? (1-9)')
```

```
 80.            move = input()
 81.        return int(move)
 82.
 83. def chooseRandomMoveFromList(board, movesList):
 84.        # Returns a valid move from the passed list on the passed board.
 85.        # Returns None if there is no valid move.
 86.        possibleMoves = []
 87.        for i in movesList:
 88.            if isSpaceFree(board, i):
 89.                possibleMoves.append(i)
 90.
 91.        if len(possibleMoves) != 0:
 92.            return random.choice(possibleMoves)
 93.        else:
 94.            return None
 95.
 96. def getComputerMove(board, computerLetter):
 97.        # Given a board and the computer's letter, determine where to move and
 return that move.
 98.        if computerLetter == 'X':
 99.            playerLetter = 'O'
100.        else:
101.            playerLetter = 'X'
102.
103.        # Here is our algorithm for our Tic Tac Toe AI:
104.        # First, check if we can win in the next move
105.        for i in range(1, 10):
106.            copy = getBoardCopy(board)
107.            if isSpaceFree(copy, i):
108.                makeMove(copy, computerLetter, i)
109.                if isWinner(copy, computerLetter):
110.                    return i
111.
112.        # Check if the player could win on their next move, and block them.
113.        for i in range(1, 10):
114.            copy = getBoardCopy(board)
115.            if isSpaceFree(copy, i):
116.                makeMove(copy, playerLetter, i)
117.                if isWinner(copy, playerLetter):
118.                    return i
119.
120.        # Try to take one of the corners, if they are free.
121.        move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
122.        if move != None:
123.            return move
124.
125.        # Try to take the center, if it is free.
```

```
126.        if isSpaceFree(board, 5):
127.            return 5
128.
129.        # Move on one of the sides.
130.        return chooseRandomMoveFromList(board, [2, 4, 6, 8])
131.
132. def isBoardFull(board):
133.     # Return True if every space on the board has been taken. Otherwise
return False.
134.     for i in range(1, 10):
135.         if isSpaceFree(board, i):
136.             return False
137.     return True
138.
139.
140. print('Welcome to Tic Tac Toe!')
141.
142. while True:
143.     # Reset the board
144.     theBoard = [' '] * 10
145.     playerLetter, computerLetter = inputPlayerLetter()
146.     turn = whoGoesFirst()
147.     print('The ' + turn + ' will go first.')
148.     gameIsPlaying = True
149.
150.     while gameIsPlaying:
151.         if turn == 'player':
152.             # Player's turn.
153.             drawBoard(theBoard)
154.             move = getPlayerMove(theBoard)
155.             makeMove(theBoard, playerLetter, move)
156.
157.             if isWinner(theBoard, playerLetter):
158.                 drawBoard(theBoard)
159.                 print('Hooray! You have won the game!')
160.                 gameIsPlaying = False
161.             else:
162.                 if isBoardFull(theBoard):
163.                     drawBoard(theBoard)
164.                     print('The game is a tie!')
165.                     break
166.                 else:
167.                     turn = 'computer'
168.
169.         else:
170.             # Computer's turn.
171.             move = getComputerMove(theBoard, computerLetter)
```

```
172.                 makeMove(theBoard, computerLetter, move)
173.
174.             if isWinner(theBoard, computerLetter):
175.                 drawBoard(theBoard)
176.                 print('The computer has beaten you! You lose.')
177.                 gameIsPlaying = False
178.             else:
179.                 if isBoardFull(theBoard):
180.                     drawBoard(theBoard)
181.                     print('The game is a tie!')
182.                     break
183.                 else:
184.                     turn = 'player'
185.
186.     if not playAgain():
187.         break
```

Designing the Program

Figure 10-1 is what a flow chart of Tic Tac Toe could look like. In the Tic Tac Toe computer program the player chooses if they want to be X or O. Who takes the first turn is randomly chosen. Then the player and computer take turns making moves.

The boxes on the left side of the flow chart are what happens during the player's turn. The right side shows what happens on the computer's turn. After the player or computer makes a move, the program checks if they won or caused a tie, and then the game switches turns. After the game is over, the program asks the player if they want to play again.
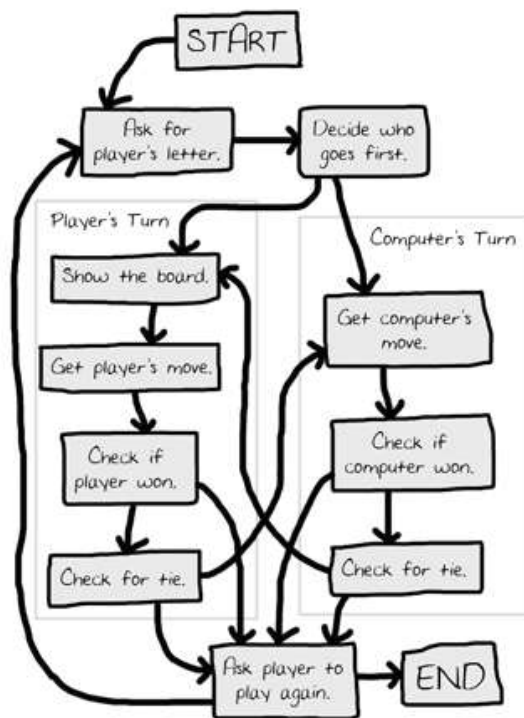
Figure 10-1: Flow chart for Tic Tac Toe
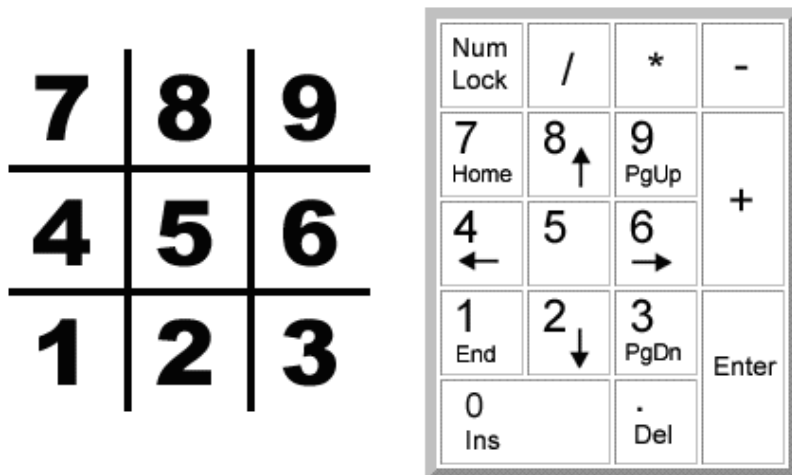
Figure 10-2: The board is numbered like the keyboard's number pad.

Representing the Board as Data

First, you must figure out how to represent the board as data in a variable. On paper, the Tic Tac Toe board is drawn as a pair of horizontal lines and a pair of vertical lines, with either an X, O, or empty space in each of the nine spaces.

In the program, the Tic Tac Toe board is represented as a list of strings. Each string will represent one of the nine spaces on the board. To make it easier to remember which index in the list is for which space, they will mirror the numbers on a keyboard's number keypad, as shown in Figure 10-2.

The strings will either be `'X'` for the X player, `'0'` for the O player, or a single space `' '` for a blank space.

So if a list with ten strings was stored in a variable named `board`, then `board[7]` would be the top-left space on the board. `board[5]` would be the center. `board[4]` would be the left side space, and so on. The program will ignore the string at index `0` in the list. The player will enter a number from 1 to 9 to tell the game which space they want to move on.

Game AI

The AI needs to be able to look at a board and decide which types of spaces it will move on. To be clear, we will label three types of spaces on the Tic Tac Toe board: corners, sides, and the center. Figure 10-3 is a chart of what each space is.
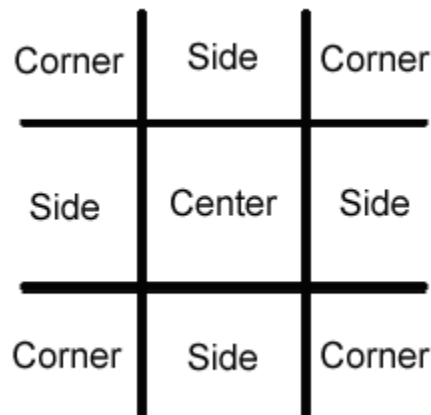
Figure 10-3: Locations of the side, corner, and center places.

The AI's smarts for playing Tic Tac Toe will follow a simple algorithm. An **algorithm** is a finite series of instructions to compute a result. A single program can make use of several different algorithms. An algorithm can be represented with a flow chart. The Tic Tac Toe AI's algorithm will compute the best move to make, as shown in Figure 10-4.

The AI's algorithm will have the following steps:

1. First, see if there's a move the computer can make that will win the game. If there is, take that move. Otherwise, go to step 2.
2. See if there's a move the player can make that will cause the computer to lose the game. If there is, move there to block the player. Otherwise, go to step 3.
3. Check if any of the corner spaces (spaces 1, 3, 7, or 9) are free. If so, move there. If no corner piece is free, then go to step 4.
4. Check if the center is free. If so, move there. If it isn't, then go to step 5.
5. Move on any of the side pieces (spaces 2, 4, 6, or 8). There are no more steps, because if the execution reaches step 5 the side spaces are the only spaces left.

This all takes place in the "Get computer's move." box on the flow chart in Figure 10-1. You could add this information to the flow chart with the boxes in Figure 10-4.
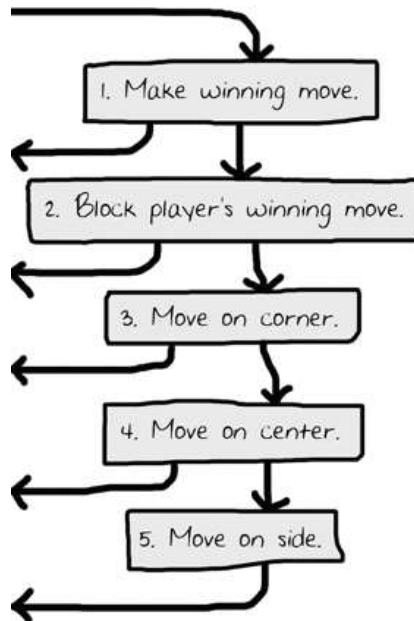
Figure 10-4: The five steps of the "Get computer's move" algorithm. The arrows leaving go to the "Check if computer won" box.

This algorithm is implemented in the `getComputerMove()` function and the other functions that `getComputerMove()` calls.

The Start of the Program

```
1. # Tic Tac Toe
2.
3. import random
```

The first couple of lines are a comment and importing the `random` module so you can call the `randint()` function.

Printing the Board on the Screen

```
 5. def drawBoard(board):
 6.     # This function prints out the board that it was passed.
 7.
 8.     # "board" is a list of 10 strings representing the board (ignore index
0)
 9.     print('   |   |')
10.     print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
11.     print('   |   |')
12.     print('-----------')
13.     print('   |   |')
14.     print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
15.     print('   |   |')
16.     print('-----------')
17.     print('   |   |')
18.     print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
19.     print('   |   |')
```

The `drawBoard()` function will print the game board represented by the `board` parameter. Remember that the board is represented as a list of ten strings, where the string at index 1 is the mark on space 1 on the Tic Tac Toe board, and so on. The string at index 0 is ignored. Many of the game's functions will work by passing a list of ten strings as the board.

Be sure to get the spacing right in the strings, otherwise the board will look funny when printed on the screen. Here are some example calls (with an argument for `board`) to `drawBoard()` and what the function would print:

```
>>> drawBoard([' ', ' ', ' ', ' ', 'X', 'O', ' ', 'X', ' ', 'O']
   |   |
 X |   | O
   |   |
```

```
 ----------
    |   |
  X | O |
    |   |
 ----------
    |   |
    |   |
    |   |
>>> [' ', 'O', 'O', ' ', ' ', 'X', ' ', ' ', ' ', ' ']
    |   |
    |   |
    |   |
 ----------
    |   |
    | X |
    |   |
 ----------
    |   |
  O | O |
    |   |
>>> [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
    |   |
    |   |
    |   |
 ----------
    |   |
    |   |
    |   |
 ----------
    |   |
    |   |
    |   |
```

Letting the Player be X or O

```
21. def inputPlayerLetter():
22.     # Lets the player type which letter they want to be.
23.     # Returns a list with the player's letter as the first item, and the
computer's letter as the second.
24.     letter = ''
25.     while not (letter == 'X' or letter == 'O'):
26.         print('Do you want to be X or O?')
27.         letter = input().upper()
```

The `inputPlayerLetter()` function asks if the player wants to be X or O. It will keep asking the player until the player types in an X or O. Line 27 automatically changes the string returned by the call to `input()` to uppercase letters with the `upper()` string method.

The `while` loop's condition contains parentheses, which means the expression inside the parentheses is evaluated first. If the `letter` variable was set to `'X'`, the expression would evaluate like this:

```
not (letter == 'X' or letter == 'O')
        ▼
not ('X' == 'X'    or 'X' == 'O')
        ▼
not (   True       or    False)
        ▼
not (True)
        ▼
not True
        ▼
    False
```

If `letter` has the value `'X'` or `'O'`, then the loop's condition is `False` and lets the program execution continue past the while-block.

```
29.      # the first element in the list is the player's letter, the second is
the computer's letter.
30.     if letter == 'X':
31.         return ['X', 'O']
32.     else:
33.         return ['O', 'X']
```

This function returns a list with two items. The first item (the string at index 0) is the player's letter, and the second item (the string at index 1) is the computer's letter. These `if-else` statements chooses the appropriate list to return.

Deciding Who Goes First

```
35. def whoGoesFirst():
36.     # Randomly choose the player who goes first.
37.     if random.randint(0, 1) == 0:
38.         return 'computer'
39.     else:
40.         return 'player'
```

The whoGoesFirst() function does a virtual coin flip to determine whether the computer or the player goes first. The coin flip is in calling random.randint(0, 1). If this function call returns a 0, the whoGoesFirst() function returns the string 'computer'. Otherwise, the function returns the string 'player'. The code that calls this function will use the return value to know who will make the first move of the game.

Asking the Player to Play Again

```
42. def playAgain():
43.     # This function returns True if the player wants to play again,
otherwise it returns False.
44.     print('Do you want to play again? (yes or no)')
45.     return input().lower().startswith('y')
```

The playAgain() function asks the player if they want to play another game. The function returns True if the player types in 'yes', 'YES', 'y', or anything that begins with the letter Y. For any other response, the function returns False. This function is identical to the one in the Hangman game.

Placing a Mark on the Board

```
47. def makeMove(board, letter, move):
48.     board[move] = letter
```

The makeMove() function is simple and only one line. The parameters are a list with ten strings named board, one of the player's letters (either 'X' or 'O') named letter, and a place on the board where that player wants to go (which is an integer from 1 to 9) named move.

But wait a second. This code seems to change one of the items in the board list to the value in letter. But because this code is in a function, the board parameter will be forgotten when the function returns. Shouldn't the change to board be forgotten as well?

Actually, this isn't the case. This is because lists are special when you pass them as arguments to functions. You are actually passing a reference of the list and not the list itself. Let's learn about the difference between lists and references to lists.

References

Try entering the following into the interactive shell:

```
>>> spam = 42
>>> cheese = spam
```

```
>>> spam = 100
>>> spam
100
>>> cheese
42
```

These results make sense from what you know so far. You assign 42 to the spam variable, and then assign the value in spam and to the variable cheese. When you later overwrite spam to 100, this doesn't affect the value in cheese. This is because spam and cheese are different variables that store different values.

But lists don't work this way. When you assign a list to a variable with the = sign, you are actually assigning a list reference to the variable. A **reference** is a value that points to some bit of data. Here is some code that will make this easier to understand. Type this into the interactive shell:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam
>>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

This looks odd. The code only changed the cheese list, but it seems that both the cheese and spam lists have changed. This is because the spam variable does not contain the list value itself, but rather spam contains a reference to the list as shown in Figure 10-5. The actual list itself is not contained in any variable, but rather exists outside of them.
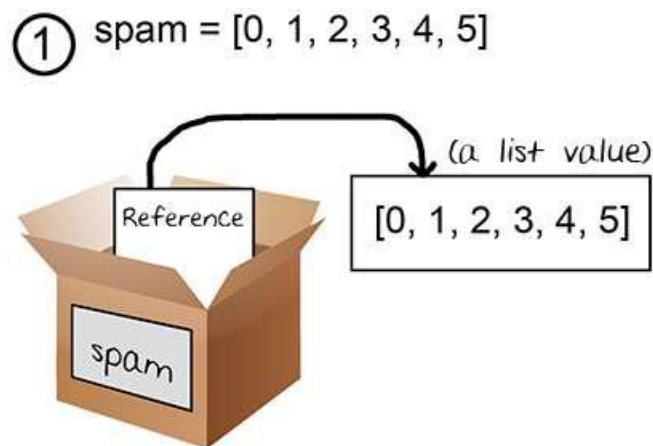
Figure 10-5: Variables don't store lists, but rather references to lists.

Notice that `cheese = spam` copies the *list reference* in `spam` to `cheese`, instead of copying the list value itself. Now both `spam` and `cheese` store a reference that refers to the same list value. But there is only one list. The list was not copied, the reference to the list was copied. Figure 10-6 shows this copying.
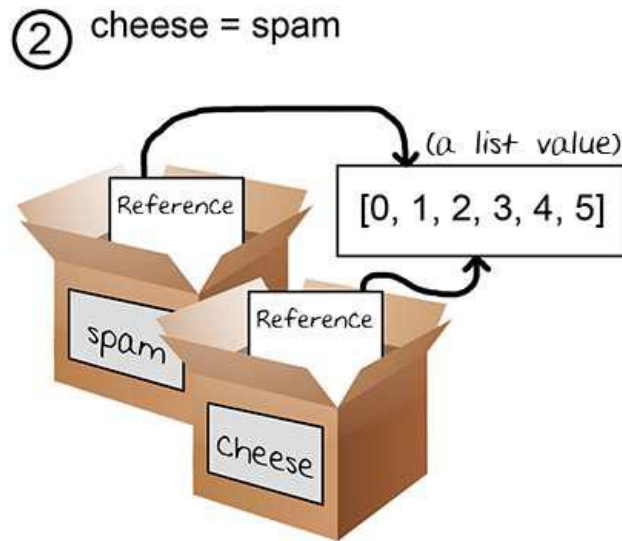
Figure 10-6: Two variables store two references to the same list.

So the `cheese[1] = 'Hello!'` line changes the same list that `spam` refers to. This is why `spam` seems to have the same list value that `cheese` does. They both have references that refer to the same list, as shown in Figure 10-7.
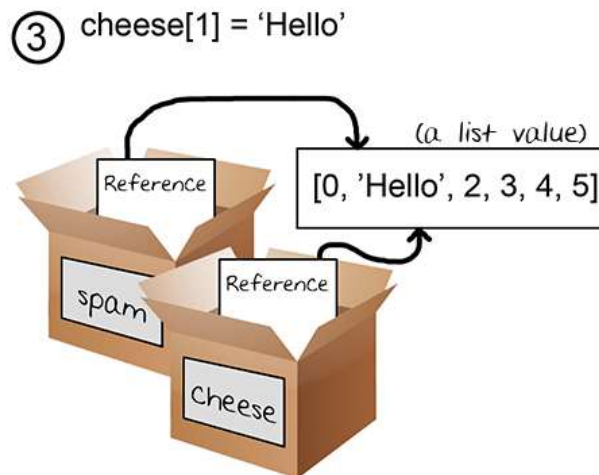
Figure 10-7: Changing the list changes all variables with references to that list.

If you want `spam` and `cheese` to store two different lists, you have to create two different lists instead of copying a reference:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = [0, 1, 2, 3, 4, 5]
```

In the above example, `spam` and `cheese` have two different lists stored in them (even though these lists are identical in content). Now if you modify one of the lists, it won't affect the other because `spam` and `cheese` have references to two different lists:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = [0, 1, 2, 3, 4, 5]
>>> cheese[1] = 'Hello!'
>>> spam
[0, 1, 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

Figure 10-8 shows how the two references point to two different lists.
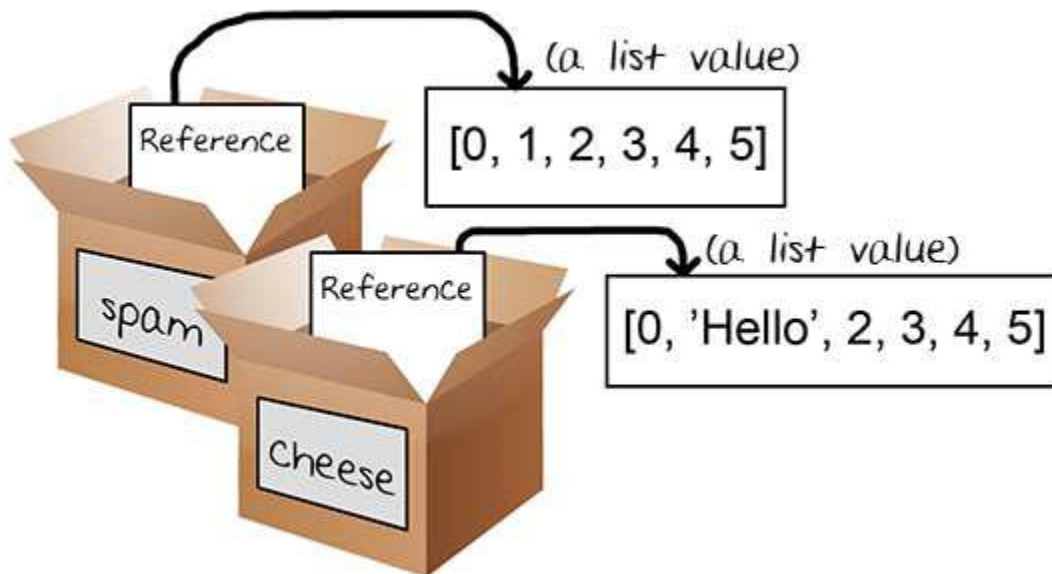
Figure 10-8: Two variables each storing references to two different lists.

Dictionaries also work the same way. Variables don't store dictionaries, they store references to dictionaries.

Using List References in makeMove()

Let's go back to the makeMove() function:

```
47. def makeMove(board, letter, move):
48.     board[move] = letter
```

When a list value is passed for the board parameter, the function's local variable is really a copy of the reference to the list, not a copy of the list. But a copy of the reference still refers to the same list the original reference refers. So any changes to board in this function will also happen to the original list. Even though board is a local variable, the makeMove() function modifies the original list.

The letter and move parameters are copies of the string and integer values that you pass. Since they are copies of values, if you modify letter or move in this function, the original variables you used when you called makeMove() aren't modified.

Checking if the Player Has Won

```
50. def isWinner(bo, le):
51.     # Given a board and a player's letter, this function returns True if
that player has won.
52.     # We use bo instead of board and le instead of letter so we don't have
to type as much.
53.     return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the
top
54.     (bo[4] == le and bo[5] == le and bo[6] == le) or # across the middle
55.     (bo[1] == le and bo[2] == le and bo[3] == le) or # across the bottom
56.     (bo[7] == le and bo[4] == le and bo[1] == le) or # down the left side
57.     (bo[8] == le and bo[5] == le and bo[2] == le) or # down the middle
58.     (bo[9] == le and bo[6] == le and bo[3] == le) or # down the right side
59.     (bo[7] == le and bo[5] == le and bo[3] == le) or # diagonal
60.     (bo[9] == le and bo[5] == le and bo[1] == le)) # diagonal
```

Lines 53 to 60 in the isWinner() function are actually one long return statement. The bo and le names are shortcuts for the board and letter parameters. These shorter names mean you have less to type in this function. Remember, Python doesn't care what you name your variables.

There are eight possible ways to win at Tic Tac Toe. You can have a line across the top, middle, and bottom rows. Or you can have a line down the left, middle, or right columns. Or you can have a line over either of the two diagonals.

Note that each line of the condition checks if the three spaces are equal to the letter provided (combined with the `and` operator) and you use the `or` operator to combine the eight different ways to win. This means only one of the eight ways must be true in order for us to say that the player who owns letter in `le` is the winner.

Let's pretend that `le` is `'O'` and `bo` is `[' ', 'O', 'O', 'O', ' ', 'X', ' ', 'X', ' ', ' ']`. The board looks like this:

```
   |   |
 X |   |
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
 O | O | O
   |   |
```

Here is how the expression after the `return` keyword on line 53 would evaluate:

```
53.       return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the
top
54.       (bo[4] == le and bo[5] == le and bo[6] == le) or # across the middle
55.       (bo[1] == le and bo[2] == le and bo[3] == le) or # across the bottom
56.       (bo[7] == le and bo[4] == le and bo[1] == le) or # down the left side
57.       (bo[8] == le and bo[5] == le and bo[2] == le) or # down the middle
58.       (bo[9] == le and bo[6] == le and bo[3] == le) or # down the right side
59.       (bo[7] == le and bo[5] == le and bo[3] == le) or # diagonal
60.       (bo[9] == le and bo[5] == le and bo[1] == le)) # diagonal
```

First Python will replace the variables `bo` and `le` with the value inside of them:

```
    return (('X' == 'O' and ' ' == 'O' and ' ' == 'O') or
    (' ' == 'O' and 'X' == 'O' and ' ' == 'O') or
    ('O' == 'O' and 'O' == 'O' and 'O' == 'O') or
    ('X' == 'O' and ' ' == 'O' and 'O' == 'O') or
    (' ' == 'O' and 'X' == 'O' and 'O' == 'O') or
    (' ' == 'O' and ' ' == 'O' and 'O' == 'O') or
    ('X' == 'O' and 'X' == 'O' and 'O' == 'O') or
    (' ' == 'O' and 'X' == 'O' and 'O' == 'O'))
```

Next, Python will evaluate all those `==` comparisons inside the parentheses to a Boolean value:

```
    return ((False and False and False) or
```

```
    (False and False and False) or
    (True and True and True) or
    (False and False and True) or
    (False and False and True) or
    (False and False and True) or
    (False and False and True) or
    (False and False and True))
```

Then the Python interpreter will evaluate all those expressions inside the parentheses:

```
    return ((False) or
    (False) or
    (True) or
    (False) or
    (False) or
    (False) or
    (False) or
    (False))
```

Since now there's only one value inside the parentheses, you can get rid of them:

```
    return (False or
    False or
    True or
    False or
    False or
    False or
    False or
    False)
```

Now evaluate the expression that is connecter by all those or operators:

```
    return (True)
```

Once again, get rid of the parentheses, and you are left with one value:

```
    return True
```

So given those values for bo and le, the expression would evaluate to True. This is how the program can tell if one of the players has won the game.

Duplicating the Board Data

```
62. def getBoardCopy(board):
```

```
63.       # Make a duplicate of the board list and return it the duplicate.
64.       dupeBoard = []
65.
66.       for i in board:
67.           dupeBoard.append(i)
68.
69.       return dupeBoard
```

The getBoardCopy() function is here so that you can easily make a copy of a given 10-string list that represents a Tic Tac Toe board in the game. There are times that you'll want the AI algorithm to make temporary modifications to a temporary copy of the board without changing the original board. In that case, call this function to make a copy of the board's list. The new list is created on line 64, with the blank list brackets [].

But the list stored in dupeBoard on line 64 is just an empty list. The for loop will iterate over the board parameter, appending a copy of the string values in the original board to the duplicate board. Finally, after the loop, dupeBoard is returned. The getBoardCopy() function builds up a copy of the original board and returning a reference to this new board in dupeBoard, and not the original one in board.

Checking if a Space on the Board is Free

```
71. def isSpaceFree(board, move):
72.     # Return true if the passed move is free on the passed board.
73.     return board[move] == ' '
```

This is a simple function that, given a Tic Tac Toe board and a possible move, will return if that move is available or not. Remember that free spaces on the board lists are marked as a single space string. If the item at the space's index is not equal to, then the space is taken.

Letting the Player Enter Their Move

```
75. def getPlayerMove(board):
76.     # Let the player type in their move.
77.     move = ' '
78.     while move not in '1 2 3 4 5 6 7 8 9'.split() or not
isSpaceFree(board, int(move)):
79.         print('What is your next move? (1-9)')
80.         move = input()
81.     return int(move)
```

The `getPlayerMove()` function asks the player to enter the number for the space they want to move on. The loop makes sure the execution does not continue until the player has entered an integer from 1 to 9. It also checks that the space entered isn't already taken, given the Tic Tac Toe board passed to the function for the `board` parameter.

The two lines of code inside the `while` loop simply ask the player to enter a number from 1 to 9. The condition on line 78 is `True` if either of the expressions on the *left* or *right* side of the `or` operator is `True`.

The expression on the *left* side checks if the player's move is equal to `'1'`, `'2'`, `'3'`, and so on up to `'9'` by creating a list with these strings (with the `split()` method) and checking if `move` is in this list.

`'1 2 3 4 5 6 7 8 9'.split()` evaluates to `['1', '2', '3', '4', '5', '6', '7', '8', '9']`, but the former easier to type.

The expression on the *right* side checks if the move that the player entered is a free space on the board. It checks this by calling the `isSpaceFree()` function. Remember that `isSpaceFree()` will return `True` if the move you pass is available on the board. Note that `isSpaceFree()` expects an integer for `move`, so the `int()` function returns an integer form of `move`.

The `not` operators are added to both sides so that the condition is `True` when either of these requirements are unfulfilled. This will cause the loop to ask the player again and again until they enter a proper move.

Finally, line 81 returns the integer form of whatever move the player entered. Remember that `input()` returns strings, so the `int()` function is called to return an integer form of the string.

Short-Circuit Evaluation

You may have noticed there's a possible problem in the `getPlayerMove()` function. What if the player typed in `'Z'` or some other non-integer string? The expression `move not in '1 2 3 4 5 6 7 8 9'.split()` on the left side of `or` would return `False` as expected, and then Python would evaluate the expression on the right side of the `or` operator.

But calling `int('Z')` would cause an error. Python gives this error because the `int()` function can only take strings of number characters, like `'9'` or `'0'`, not strings like `'Z'`.

As an example of this kind of error, try entering this into the interactive shell:

```
>>> int('42')
42
>>> int('Z')
Traceback (most recent call last):
```

```
   File "<pyshell#3>", line 1, in <module>
     int('Z')
ValueError: invalid literal for int() with base 10: 'Z'
```

But when you play the Tic Tac Toe game and try entering `'Z'` for your move, this error doesn't happen. The reason is because the `while` loop's condition is being short-circuited.

Short-circuiting means is that since the part on the left side of the `or` keyword (`move not in '1 2 3 4 5 6 7 8 9'.split()`) evaluates to `True`, the Python interpreter knows that the entire expression will evaluate to `True`. It doesn't matter if the expression on the right side of the `or` keyword evaluates to `True` or `False`, because only one value on the side of the `or` operator needs to be `True`.

Think about it: The expression `True or False` evaluates to `True` and the expression `True or True` also evaluates to `True`. If the value on the left side is `True`, it doesn't matter what the value is on the right side:

<div align="center">

`False and <<<anything>>>` always evaluates to `False`

`True or <<<anything>>>` always evaluates to `True`

</div>

So Python stops checking the rest of the expression and doesn't even bother evaluating the `not isSpaceFree(board, int(move))` part. This means the `int()` and the `isSpaceFree()` functions are never called as long as `move not in '1 2 3 4 5 6 7 8 9'.split()` is `True`.

This works out well for the program, because if the right side is `True` then `move` isn't a string in number form. That would cause `int()` to give us an error. The only times `move not in '1 2 3 4 5 6 7 8 9'.split()` evaluates to `False` are when `move` isn't a single-digit string. In that case, the call to `int()` would not give us an error.

An Example of Short-Circuit Evaluation

Here's a short program that gives a good example of short-circuiting. Try entering the following into the interactive shell:

```
>>> def ReturnsTrue():
        print('ReturnsTrue() was called.')
        return True

>>> def ReturnsFalse():
        print('ReturnsFalse() was called.')
        return False

>>> ReturnsTrue()
ReturnsTrue() was called.
```

```
True
>>> ReturnsFalse()
ReturnsFalse() was called.
False
```

When `ReturnsTrue()` is called, it prints `'ReturnsTrue() was called.'` and then also displays the return value of `ReturnsTrue()`. The same goes for `ReturnsFalse()`.

Now try entering the following into the interactive shell.

```
>>> ReturnsFalse() or ReturnsTrue()
ReturnsFalse() was called.
ReturnsTrue() was called.
True
>>> ReturnsTrue() or ReturnsFalse()
ReturnsTrue() was called.
True
```

The first part makes sense: The expression `ReturnsFalse() or ReturnsTrue()` calls both of the functions, so you see both of the printed messages.

But the second expression only shows `'ReturnsTrue() was called.'` but not `'ReturnsFalse() was called.'`. This is because Python did not call `ReturnsFalse()` at all. Since the left side of the `or` operator is `True`, it doesn't matter what `ReturnsFalse()` returns and Python doesn't bother calling it. The evaluation was short-circuited.

The same applies for the `and` operator. Try entering the following into the interactive shell:

```
>>> ReturnsTrue() and ReturnsTrue()
ReturnsTrue() was called.
ReturnsTrue() was called.
True
>>> ReturnsFalse() and ReturnsFalse()
ReturnsFalse() was called.
False
```

If the left side of the `and` operator is `False`, then the entire expression is `False`. It doesn't matter whether the right side of the `and` operator is `True` or `False`, so Python doesn't bother evaluating it. Both `False and True` and `False and False` evaluate to `False`, so Python short-circuits the evaluation.

Choosing a Move from a List of Moves

```
83. def chooseRandomMoveFromList(board, movesList):
84.     # Returns a valid move from the passed list on the passed board.
```

```
85.        # Returns None if there is no valid move.
86.        possibleMoves = []
87.        for i in movesList:
88.            if isSpaceFree(board, i):
89.                possibleMoves.append(i)
```

The chooseRandomMoveFromList() function is useful for the AI code later in the program. The board parameter is a list of strings that represents a Tic Tac Toe board. The second parameter movesList is a list of integers of possible spaces from which to choose. For example, if movesList is [1, 3, 7, 9], that means chooseRandomMoveFromList() should return the integer for one of the corner spaces.

However, chooseRandomMoveFromList() will first check that the space is valid to make a move on. The possibleMoves list starts as a blank list. The for loop will iterate over movesList. The moves that cause isSpaceFree() to return True are added to possibleMoves with the append() method.

```
91.        if len(possibleMoves) != 0:
92.            return random.choice(possibleMoves)
93.        else:
94.            return None
```

At this point, the possibleMoves list has all of the moves that were in movesList that are also free spaces. If the list isn't empty, then there's at least one possible move that can be made on the board.

But this list could be empty. For example, if movesList was [1, 3, 7, 9] but the board represented by the board parameter had all the corner spaces already taken, the possibleMoves list would be []. In that case, len(possibleMoves) will evaluate to 0 and the function returns the value None. This next section explains the None value.

The None Value

The **None value** is a value that represents the lack of a value. None is the only value of the data type NoneType. It can be useful to use the None value when you need a value that means "does not exist" or "none of the above".

For example, say you had a variable named quizAnswer which holds the user's answer to some True-False pop quiz question. The variable could hold True or False for the user's answer. You could set quizAnswer to None if the user skipped the question and didn't answer it. Using None would be better because otherwise it may look like the user answered the question when they didn't.

Functions that return by reaching the end of the function (and not from a `return` statement) have `None` for a return value. The `None` value is written without quotes and with a capital "N" and lowercase "one".

As a side note, `None` will not be displayed in the interactive shell like other values will be:

```
>>> 2 + 2
4
>>> 'This is a string value.'
'This is a string value.'
>>> None
```

Functions that don't seem to return anything actually return the `None` value. For example, `print()` returns `None`:

```
>>> spam = print('Hello world!')
Hello world!
>>> spam == None
True
```

Creating the Computer's Artificial Intelligence

```
 96. def getComputerMove(board, computerLetter):
 97.     # Given a board and the computer's letter, determine where to move and
return that move.
 98.     if computerLetter == 'X':
 99.         playerLetter = 'O'
100.     else:
101.         playerLetter = 'X'
```

The `getComputerMove()` function contains the AI's code. The first argument is a Tic Tac Toe board for the `board` parameter. The second argument is letter for the computer either `'X'` or `'O'` in the `computerLetter` parameter. The first few lines simply assign the other letter to a variable named `playerLetter`. This way the same code can be used whether the computer is X or O.

The function will returns an integer from `1` to `9` representing the computer's move.

Remember how the Tic Tac Toe AI algorithm works:

- First, see if there's a move the computer can make that will win the game. If there is, take that move. Otherwise, go to the second step.

- Second, see if there's a move the player can make that will cause the computer to lose the game. If there is, the computer should move there to block the player. Otherwise, go to the third step.

- Third, check if any of the corner spaces (spaces 1, 3, 7, or 9) are free. If no corner space is free, then go to the fourth step.

- Fourth, check if the center is free. If so, move there. If it isn't, then go to the fifth step.

- Fifth, move on any of the side pieces (spaces 2, 4, 6, or 8). There are no more steps, because if the execution has reached this step then the side spaces are the only spaces left.

The Computer Checks if it Can Win in One Move

```
103.      # Here is our algorithm for our Tic Tac Toe AI:
104.      # First, check if we can win in the next move
105.      for i in range(1, 10):
106.          copy = getBoardCopy(board)
107.          if isSpaceFree(copy, i):
108.              makeMove(copy, computerLetter, i)
109.              if isWinner(copy, computerLetter):
110.                  return i
```

More than anything, if the computer can win in the next move, the computer should make that winning move immediately. The `for` loop that starts on line 105 iterates over every possible move from 1 to 9. The code inside the loop will simulate what would happen if the computer made that move.

The first line in the loop (line 106) makes a copy of the `board` list. This is so the simulated move inside the loop doesn't modify the real Tic Tac Toe board stored in the `board` variable. The `getBoardCopy()` returns an identical but separate board list value.

Line 107 checks if the space is free and if so, simulates making the move on the copy of the board. If this move results in the computer winning, the function returns that move's integer.

If none of the spaces results in winning, the loop will finally end and the program execution continues to line 113.

The Computer Checks if the Player Can Win in One Move

```
112.      # Check if the player could win on their next move, and block them.
113.      for i in range(1, 10):
114.          copy = getBoardCopy(board)
115.          if isSpaceFree(copy, i):
```

```
116.                makeMove(copy, playerLetter, i)
117.                if isWinner(copy, playerLetter):
118.                    return i
```

Next, the code will simulate the human player moving on each of the spaces. The code is similar to the previous loop except the player's letter is put on the board copy. If the isWinner() function shows that the player would win with this move, then the computer will return that same move to block this from happening.

If the human player cannot win in one more move, the for loop will eventually finish and execution continues to line 121.

Checking the Corner, Center, and Side Spaces (in that Order)

```
120.        # Try to take one of the corners, if they are free.
121.        move = chooseRandomMoveFromList(board, [1, 3, 7, 9])
122.        if move != None:
123.            return move
```

The call to chooseRandomMoveFromList() with the list of [1, 3, 7, 9] will ensure that it returns the integer for one of the corner spaces: 1, 3, 7, or 9. If all the corner spaces are taken, the chooseRandomMoveFromList() function will return None and execution moves on to line 126.

```
125.        # Try to take the center, if it is free.
126.        if isSpaceFree(board, 5):
127.            return 5
```

If none of the corners are available, line 127 moves on the center space if it is free. If the center space isn't free, the execution moves on to line 130.

```
129.        # Move on one of the sides.
130.        return chooseRandomMoveFromList(board, [2, 4, 6, 8])
```

This code also makes a call to chooseRandomMoveFromList(), except you pass it a list of the side spaces ([2, 4, 6, 8]). This function won't return None because the side spaces are the only spaces that can possibly be left. This ends the getComputerMove() function and the AI algorithm.

Checking if the Board is Full

```
132. def isBoardFull(board):
133.     # Return True if every space on the board has been taken. Otherwise
return False.
```

```
134.        for i in range(1, 10):
135.            if isSpaceFree(board, i):
136.                return False
137.        return True
```

The last function is `isBoardFull()`. This function returns `True` if the 10-string list `board` argument it was passed has an `'X'` or `'O'` in every index (except for index `0`, which is ignored). If there's at least one space in `board` that is set to a single space `' '` then it will return `False`.

The `for` loop will let us check indexes `1` through `9` on the `board` list. As soon as it finds a free space on the board (that is, when `isSpaceFree(board, i)` returns `True`) the `isBoardFull()` function will return `False`.

If execution manages to go through every iteration of the loop, then none of the spaces are free. Line 137 will then execute `return True`.

The Start of the Game

```
140. print('Welcome to Tic Tac Toe!')
```

Line 140 is the first line that isn't inside of a function, so it is the first line of code that executes when you run this program. It greets the player.

```
142. while True:
143.     # Reset the board
144.     theBoard = [' '] * 10
```

Line 142's `while` loop has `True` for the condition and will keep looping until the execution encounters a `break` statement. Line 144 sets up the main Tic Tac Toe board in a variable named `theBoard`. It is a 10-string list, where each string is a single space `' '`.

Rather than type out this full list, line 144 uses list replication. It is shorter to type `[' '] * 10` then `[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']`.

Deciding the Player's Mark and Who Goes First

```
145.     playerLetter, computerLetter = inputPlayerLetter()
```

The `inputPlayerLetter()` function lets the player type in whether they want to be X or O. The function returns a 2-string list, either `['X', 'O']` or `['O', 'X']`. The multiple assignment trick will set `playerLetter` to the first item in the returned list and `computerLetter` to the second.

```
146.        turn = whoGoesFirst()
147.        print('The ' + turn + ' will go first.')
148.        gameIsPlaying = True
```

The `whoGoesFirst()` function randomly decides who goes first, and returns either the string `'player'` or the string `'computer'` and line 147 tells the player who will go first. The `gameIsPlayer` variable keeps track of whether the game is still being played or if someone has won or tied.

Running the Player's Turn

```
150.        while gameIsPlaying:
```

Line 150's loop will keep going back and forth between the code for the player's turn and the computer's turn, as long as `gameIsPlaying` is set to `True`.

```
151.            if turn == 'player':
152.                # Player's turn.
153.                drawBoard(theBoard)
154.                move = getPlayerMove(theBoard)
155.                makeMove(theBoard, playerLetter, move)
```

The `turn` variable was originally set by the `whoGoesFirst()` call on line 146. It is set either to `'player'` or `'computer'`. If `turn` equals `'computer'`, then line 151's condition is `False` and execution jumps to line 169.

Line 153 calls `drawBoard()` and passes the `theBoard` variable to print the Tic Tac Toe board on the screen. Then the `getPlayerMove()` function lets the player type in their move (and also makes sure it is a valid move). The `makeMove()` function adds the player's X or O to `theBoard`.

```
157.                if isWinner(theBoard, playerLetter):
158.                    drawBoard(theBoard)
159.                    print('Hooray! You have won the game!')
160.                    gameIsPlaying = False
```

Now that the player has made their move, the computer should check if they have won the game with this move. If the `isWinner()` function returns `True`, the if-block's code displays the winning board and prints a message telling them they have won.

The `gameIsPlaying` variable is also set to `False` so that execution doesn't continue on to the computer's turn.

```
161.                else:
162.                    if isBoardFull(theBoard):
163.                        drawBoard(theBoard)
164.                        print('The game is a tie!')
165.                        break
```

If the player didn't win with their last move, maybe their move filled up the entire board and tied the game. In this else-block, the `isBoardFull()` function returns `True` if there are no more moves to make. In that case, the if-block starting at line 162 displays the tied board and tell the player a tie has occurred. The execution breaks out of the `while` loop and jumps to line 186.

```
166.                    else:
167.                        turn = 'computer'
```

If the player hasn't won or tied the game, then line 167 sets the `turn` variable to `'computer'` so that it will execute the code for the computer's turn on the next iteration.

Running the Computer's Turn

If the `turn` variable wasn't `'player'` for the condition on line 151, then it must be the computer's turn. The code in this else-block is similar to the code for the player's turn.

```
169.        else:
170.            # Computer's turn.
171.            move = getComputerMove(theBoard, computerLetter)
172.            makeMove(theBoard, computerLetter, move)
174.            if isWinner(theBoard, computerLetter):
175.                drawBoard(theBoard)
176.                print('The computer has beaten you! You lose.')
177.                gameIsPlaying = False
178.            else:
179.                if isBoardFull(theBoard):
180.                    drawBoard(theBoard)
181.                    print('The game is a tie!')
182.                    break
183.                else:
184.                    turn = 'player'
```

Lines 170 to 184 are almost identical to the code for the player's turn on lines 152 to 167. The only difference is this the code uses the computer's letter and calls `getComputerMove()`.

If the game isn't won or tied, line 184 sets `turn` to the player's turn. There are no more lines of code inside the `while` loop, so execution would jump back to the `while` statement on line 150.

```
186.        if not playAgain():
187.            break
```

Lines 186 and 187 are located immediately after the while-block started by the `while` statement on line 150. `gameIsPlaying` is set to `False` when the game has ended, so at this point the game asks the player if they want to play again.

If `playAgain()` returns `False`, then the `if` statement's condition is `True` (because the `not` operator reverses the Boolean value) and the `break` statement executes. That breaks the execution out of the `while` loop that was started on line 142. But since there are no more lines of code after that while-block, the program terminates.

Summary

Creating a program that can play a game comes down to carefully considering all the possible situations the AI can be in and how it should respond in each of those situations. The Tic Tac Toe AI is simple because there are not many possible moves in Tic Tac Toe compared to a game like chess or checkers.

Our AI checks if any possible move can allow itself to win. Otherwise, it checks if it must block the player's move. Then the AI simply chooses any available corner space, then the center space, then the side spaces. This is a simple algorithm for the computer to follow.

The key to implementing our AI is by making copies of the board data and simulating moves on the copy. That way, the AI code can see if a move results in a win or loss. Then the AI can make that move on the real board. This type of simulation is effective at predicting what is a good move or not.

Chapter 11

BAGELS

> Topics Covered In This Chapter:
> - Augmented Assignment Operators, +=, -=, *=, /=
> - The `random.shuffle()` Function
> - The `sort()` and `join()` List Methods
> - String Interpolation (also called String Formatting)
> - Conversion Specifier `%s`
> - Nested Loops

In this chapter, you'll learn a few new methods and functions that come with Python. You'll also learn about augmented assignment operators and string interpolation. These things don't let you do anything you couldn't do before, but they are nice shortcuts to make coding easier.

Bagels is a deduction game you can play with a friend. Your friend thinks up a random 3-digit number with no repeating digits, and you try to guess what the number is. After each guess, your friend gives you three types of clues:

- **Bagels** – None of the three digits you guessed is in the secret number.

- **Pico** – One of the digits is in the secret number, but your guess has the digit in the wrong place.

- **Fermi** – Your guess has a correct digit in the correct place.

You can get multiple clues after each guess. If the secret number is 456 and your guess is 546 the clues would be "fermi pico pico". The 6 provides "fermi" and the 5 and 4 provide "pico pico".

Sample Run of Bagels

```
I am thinking of a 3-digit number. Try to guess what it is.
Here are some clues:
When I say:    That means:
  Pico          One digit is correct but in the wrong position.
  Fermi         One digit is correct and in the right position.
  Bagels        No digit is correct.
I have thought up a number. You have 10 guesses to get it.
Guess #1:
123
```

```
Fermi
Guess #2:
453
Pico
Guess #3:
425
Fermi
Guess #4:
326
Bagels
Guess #5:
489
Bagels
Guess #6:
075
Fermi Fermi
Guess #7:
015
Fermi Pico
Guess #8:
175
You got it!
Do you want to play again? (yes or no)
no
```

Source Code of Bagels

If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/bagels.

bagels.py

```python
1. import random
2. def getSecretNum(numDigits):
3.     # Returns a string that is numDigits long, made up of unique random
digits.
4.     numbers = list(range(10))
5.     random.shuffle(numbers)
6.     secretNum = ''
7.     for i in range(numDigits):
8.         secretNum += str(numbers[i])
9.     return secretNum
10.
11. def getClues(guess, secretNum):
12.     # Returns a string with the pico, fermi, bagels clues to the user.
13.     if guess == secretNum:
```

```
14.            return 'You got it!'
15.
16.        clue = []
17.
18.        for i in range(len(guess)):
19.            if guess[i] == secretNum[i]:
20.                clue.append('Fermi')
21.            elif guess[i] in secretNum:
22.                clue.append('Pico')
23.        if len(clue) == 0:
24.            return 'Bagels'
25.
26.        clue.sort()
27.        return ' '.join(clue)
28.
29. def isOnlyDigits(num):
30.        # Returns True if num is a string made up only of digits. Otherwise
returns False.
31.        if num == '':
32.            return False
33.
34.        for i in num:
35.            if i not in '0 1 2 3 4 5 6 7 8 9'.split():
36.                return False
37.
38.        return True
39.
40. def playAgain():
41.        # This function returns True if the player wants to play again,
otherwise it returns False.
42.        print('Do you want to play again? (yes or no)')
43.        return input().lower().startswith('y')
44.
45. NUMDIGITS = 3
46. MAXGUESS = 10
47.
48. print('I am thinking of a %s-digit number. Try to guess what it is.' %
(NUMDIGITS))
49. print('Here are some clues:')
50. print('When I say:     That means:')
51. print('  Pico          One digit is correct but in the wrong position.')
52. print('  Fermi         One digit is correct and in the right position.')
53. print('  Bagels        No digit is correct.')
54.
55. while True:
56.        secretNum = getSecretNum(NUMDIGITS)
```

```
57.     print('I have thought up a number. You have %s guesses to get it.' %
(MAXGUESS))
58.
59.     numGuesses = 1
60.     while numGuesses <= MAXGUESS:
61.         guess = ''
62.         while len(guess) != NUMDIGITS or not isOnlyDigits(guess):
63.             print('Guess #%s: ' % (numGuesses))
64.             guess = input()
65.
66.         clue = getClues(guess, secretNum)
67.         print(clue)
68.         numGuesses += 1
69.
70.         if guess == secretNum:
71.             break
72.         if numGuesses > MAXGUESS:
73.             print('You ran out of guesses. The answer was %s.' %
(secretNum))
74.
75.     if not playAgain():
76.         break
```

Designing the Program

The flow chart in Figure 11-1 describes what happens in this game, and in what order they can happen.

How the Code Works

```
1. import random
2. def getSecretNum(numDigits):
3.     # Returns a string that is numDigits long, made up of unique random
digits.
```

At the start of the program, import the random module. Then define a function named getSecretNum(). The function makes a secret number that has only unique digits in it. Instead of only 3-digit secret numbers, the numDigits parameter lets the function make a secret number with any number of digits. For example, you can make a secret number of four or six digits by passing 4 or 6 for numDigits.
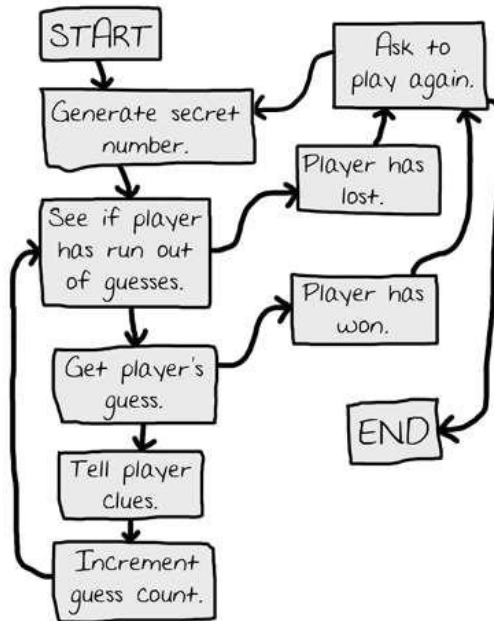
Figure 11-1: Flow chart for the Bagels game.

Shuffling a Unique Set of Digits

```
4.        numbers = list(range(10))
5.        random.shuffle(numbers)
```

Line 4's list(range(10)) always evaluate to [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. It's just easier to type list(range(10)). The numbers variable contains a list of all ten digits.

The `random.shuffle()` Function

The random.shuffle() function randomly changes the order of a list's items. This function doesn't return a value, but rather modifies the list you pass it "in place". This is similar to the way the makeMove() function in the Tic Tac Toe chapter modified the list it was passed in place, rather than return a new list with the change. This is why you do **not** write code like numbers = random.shuffle(numbers).

Try experimenting with the random.shuffle() function by entering the following code into the interactive shell:

```
>>> import random
>>> spam = list(range(10))
>>> print(spam)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> random.shuffle(spam)
>>> print(spam)
[3, 0, 5, 9, 6, 8, 2, 4, 1, 7]

>>> random.shuffle(spam)
>>> print(spam)
[1, 2, 5, 9, 4, 7, 0, 3, 6, 8]

>>> random.shuffle(spam)
>>> print(spam)
[9, 8, 3, 5, 4, 7, 1, 2, 0, 6]
```

You want the secret number in Bagels to have unique digits. The Bagels game is much more fun if you don't have duplicate digits in the secret number, such as '244' or '333'. The shuffle() function will help you do this.

Getting the Secret Number from the Shuffled Digits

```
6.        secretNum = ''
7.        for i in range(numDigits):
8.            secretNum += str(numbers[i])
9.        return secretNum
```

The secret number will be a string of the first numDigits digits of the shuffled list of integers. For example, if the shuffled list in numbers is [9, 8, 3, 5, 4, 7, 1, 2, 0, 6] and numDigits was 3, then you'd want the string returned by getSecretNum() to be '983'.

To do this, the secretNum variable starts out as a blank string. The for loop on line 7 iterates numDigits number of times. On each iteration through the loop, the integer at index i is pulled from the shuffled list, converted to a string, and concatenated to the end of secretNum.

For example, if numbers refers to the list [9, 8, 3, 5, 4, 7, 1, 2, 0, 6], then on the first iteration, numbers[0] (that is, 9) will be passed to str(), which in turn returns '9' which is concatenated to the end of secretNum. On the second iteration, the same happens with numbers[1] (that is, 8) and on the third iteration the same happens with numbers[2] (that is, 3). The final value of secretNum that is returned is '983'.

Notice that secretNum in this function contains a string, not an integer. This may seem odd, but remember that you cannot concatenate integers. The expression 9 + 8 + 3 evaluates to 20, but what you want is '9' + '8' + '3', which evaluates to '983'.

Augmented Assignment Operators

The += operator on line 8 is one of the **augmented assignment operators**. Normally, if you wanted to add or concatenate a value to a variable, you would use code that looked like this:

```
spam = 42
spam = spam + 10

eggs = 'Hello '
eggs = eggs + 'world!'
```

The augmented assignment operators are a shortcut that frees you from retyping the variable name. The following code does the same thing as the above code:

```
spam = 42
spam += 10        # Like spam = spam + 10

eggs = 'Hello '
eggs += 'world!' # Like eggs = eggs + 'world!'
```

There are other augmented assignment operators as well. Try entering the following into the interactive shell:

```
>>> spam = 42
>>> spam -= 2
>>> spam
40
>>> spam *= 3
>>> spam
120
>>> spam /= 10
>>> spam
12.0
```

Calculating the Clues to Give

```
11. def getClues(guess, secretNum):
12.     # Returns a string with the pico, fermi, bagels clues to the user.
13.     if guess == secretNum:
14.         return 'You got it!'
```

The getClues() function will return a string with the fermi, pico, and bagels clues depending on the guess and secretNum parameters. The most obvious and easiest step is to check if the guess is the same as the secret number. In that case, line 14 returns 'You got it!'.

```
16.      clue = []
17.
18.      for i in range(len(guess)):
19.          if guess[i] == secretNum[i]:
20.              clue.append('Fermi')
21.          elif guess[i] in secretNum:
22.              clue.append('Pico')
```

If the guess isn't the same as the secret number, the code must figure out what clues to give the player. The list in `clue` will start empty and have `'Fermi'` and `'Pico'` strings added as needed.

Do this by looping through each possible index in `guess` and `secretNum`. The strings in both variables will be the same length, so the line 18 could have used either `len(guess)` or `len(secretNum)` and work the same. As the value of `i` changes from 0 to 1 to 2, and so on, line 19 checks if the first, second, third, etc. letter of `guess` is the same as the number in the same index of `secretNum`. If so, line 20 will add a string `'Fermi'` to clue.

Otherwise, line 21 will check if the number at the `i`th position in guess exists anywhere in `secretNum`. If so, you know that the number is somewhere in the secret number but not in the same position. Line 22 will then add `'Pico'` to `clue`.

```
23.      if len(clue) == 0:
24.          return 'Bagels'
```

If the `clue` list is empty after the loop, then you know that there are no correct digits at all in `guess`. In this case, line 24 returns the string `'Bagels'` as the only clue.

The `sort()` List Method

```
26.      clue.sort()
```

Lists have a method named `sort()` that rearranges the items in the list to be in alphabetical or numerical order. Try entering the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'bat', 'anteater']
>>> spam.sort()
>>> spam
['anteater', 'bat', 'cat', 'dog']

>>> spam = [9, 8, 3, 5, 4, 7, 1, 2, 0, 6]
>>> spam.sort()
>>> spam
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The `sort()` method doesn't return a sorted list, but rather sorts the list it is called on "in place". This is just like how the `reverse()` method works.

You would never want to use this line of code: `return spam.sort()` because that would return the value `None` (which is what `sort()` returns). Instead you would want a separate line `spam.sort()` and then the line `return spam`.

The reason you want to sort the `clue` list is to get rid of extra information based on the order of the clues. If `clue` was `['Pico', 'Fermi', 'Pico']`, then that would tell the player that the center digit of the guess is in the correct position. Since the other two clues are both Pico, the player would know that all they have to do is swap the first and third digit to get the secret number.

If the clues are always sorted in alphabetical order, the player can't be sure which number the Fermi clue refers. This is what we want for the game.

The `join()` String Method

```
27.     return ' '.join(clue)
```

The `join()` string method returns a list of strings as a single string joined together. The string that the method is called on (on line 27, this is a single space, `' '`) appears between each string in the list. For an example, enter the following into the interactive shell:

```
>>> ' '.join(['My', 'name', 'is', 'Zophie'])
'My name is Zophie'
>>> ', '.join(['Life', 'the Universe', 'and Everything'])
'Life, the Universe, and Everything'
```

So the string that is returned on line 27 is each string in `clue` combined together with a single space between each string. The `join()` string method is sort of like the opposite of the `split()` string method. While `split()` returns a list from a split up string, `join()` returns a string from a combined list.

Checking if a String Has Only Numbers

```
29. def isOnlyDigits(num):
30.     # Returns True if num is a string made up only of digits. Otherwise
returns False.
31.     if num == '':
32.         return False
```

The isOnlyDigits() helps determine if the player entered a valid guess. Line 31 checks if num is the blank string, and if so, returns False.

```
34.      for i in num:
35.          if i not in '0 1 2 3 4 5 6 7 8 9'.split():
36.              return False
37.
38.      return True
```

The for loop iterates over each character in the string num. The value of i will have a single character on each iteration. Inside the for-block, the code checks if i doesn't exist in the list returned by '0 1 2 3 4 5 6 7 8 9'.split(). (The return value from split() is equivalent to ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'] but is easier to type.) If it doesn't, you know there's a non-digit character in num. In that case, line 36 returns False.

If execution continues past the for loop, then you know that every character in num is a digit. In that case, line 38 returns True.

Finding out if the Player Wants to Play Again

```
40. def playAgain():
41.     # This function returns True if the player wants to play again,
otherwise it returns False.
42.     print('Do you want to play again? (yes or no)')
43.     return input().lower().startswith('y')
```

The playAgain() function is the same one you used in Hangman and Tic Tac Toe. The long expression on line 43 evaluates to either True or False based on the answer given by the player.

The Start of the Game

```
45. NUMDIGITS = 3
46. MAXGUESS = 10
47.
48. print('I am thinking of a %s-digit number. Try to guess what it is.' %
(NUMDIGITS))
49. print('Here are some clues:')
50. print('When I say:    That means:')
51. print('  Pico         One digit is correct but in the wrong position.')
52. print('  Fermi        One digit is correct and in the right position.')
53. print('  Bagels       No digit is correct.')
```

After all of the function definitions, this is the actual start of the program. Instead of using the integer 3 in our program for the number of answer has, use the constant variable NUMDIGITS. The same goes for using the constant variable MAXGUESS instead of the integer 10 for the number of guesses the player gets. Now it will be easy to change the number of guesses or secret number digits. Just change line 45 or 46 and the rest of the program will still work without any more changes.

The print() function calls will tell the player the rules of the game and what the Pico, Fermi, and Bagels clues mean. Line 48's print() call has % (NUMDIGITS) added to the end and %s inside the string. This is a technique known as string interpolation.

String Interpolation

String interpolation is a coding shortcut. Normally, if you want to use the string values inside variables in another string, you have to use the + concatenation operator:

```
>>> name = 'Alice'
>>> event = 'party'
>>> where = 'the pool'
>>> day = 'Saturday'
>>> time = '6:00pm'

>>> print('Hello, ' + name + '. Will you go to the ' + event + ' at ' + where +
' this ' + day + ' at ' + time + '?')
Hello, Alice. Will you go to the party at the pool this Saturday at 6:00pm?
```

As you can see, it can be hard to type a line that concatenates several strings. Instead, you can use **string interpolation**, which lets you put placeholders like %s. These placeholders are called **conversion specifiers**. Then put all the variable names at the end. Each %s is replaced with a variable at the end of the line. For example, the following code does the same thing as the previous code:

```
>>> name = 'Alice'
>>> event = 'party'
>>> where = 'the pool'
>>> day = 'Saturday'
>>> time = '6:00pm'

>>> print('Hello, %s. Will you go to the %s at %s this %s at %s?' % (name,
event, where, day, time))
Hello, Alice. Will you go to the party at the pool this Saturday at 6:00pm?
```

String interpolation can make your code much easier to type. The first variable name is used for the first %s, the second variable with the second %s and so on. You must have the same number of %s conversion specifiers as you have variables.

Another benefit of using string interpolation instead of string concatenation is interpolation works with any data type, not just strings. All values are automatically converted to the string data type. If you concatenated an integer to a string, you'd get this error:

```
>>> spam = 42
>>> print('Spam == ' + spam)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

String concatenation can only combine two strings, but spam is an integer. You would have to remember to put str(spam) instead of spam. But with string interpolation, this conversion to strings is done for you. Try entering this into the interactive shell:

```
>>> spam = 42
>>> print('Spam is %s' % (spam))
Spam is 42
```

String interpolation is also known as **string formatting**.

Creating the Secret Number

```
55. while True:
56.     secretNum = getSecretNum(NUMDIGITS)
57.     print('I have thought up a number. You have %s guesses to get it.' %
(MAXGUESS))
58.
59.     numGuesses = 1
60.     while numGuesses <= MAXGUESS:
```

Line 55 is an infinite while loop that has a condition of True so it will loop forever until a break statement is executed. Inside the infinite loop, you get a secret number from the getSecretNum() function, passing it NUMDIGITS to tell how many digits you want the secret number to have. This secret number is assigned to secretNum. Remember, the value in secretNum is a string not an integer.

Line 57 tells the player how many digits is in the secret number by using string interpolation instead of string concatenation. Line 59 sets variable numGuesses to 1 to mark this is as the first

guess. Then line 60 has a new `while` loop that loops as long as `numGuesses` is less than or equal to `MAXGUESS`.

Getting the Player's Guess

```
61.          guess = ''
62.          while len(guess) != NUMDIGITS or not isOnlyDigits(guess):
63.              print('Guess #%s: ' % (numGuesses))
64.              guess = input()
```

The `guess` variable will hold the player's guess returned from `input()`. The code keeps looping and asking the player for a guess until the player enters a valid guess. A valid guess has only digits and the same number of digits as the secret number. This is what the `while` loop that starts on line 62 is for.

The `guess` variable is set to the blank string on line 61 so the `while` loop's condition is `False` the first time it is checked, ensuring the execution enters the loop.

Getting the Clues for the Player's Guess

```
66.          clue = getClues(guess, secretNum)
67.          print(clue)
68.          numGuesses += 1
```

After execution gets past the `while` loop that started on line 62, `guess` contains a valid guess. Pass this and `secretNum` to the `getClues()` function. It returns a string of the clues, which are displayed to the player on line 67. Line 68 increments `numGuesses` using the augmented assignment operator for addition.

Checking if the Player Won or Lost

Notice that this second `while` loop on line 60 is inside another `while` loop that started on line 55. These loops-inside-loops are called **nested loops**. Any `break` or `continue` statements will only break or continue out of the innermost loop, and not any of the outer loop.

```
70.          if guess == secretNum:
71.              break
72.          if numGuesses > MAXGUESS:
73.              print('You ran out of guesses. The answer was %s.' %
(secretNum))
```

If `guess` is the same value as `secretNum`, the player has correctly guessed the secret number and line 71 breaks out of the `while` loop that was started on line 60.

If not, then execution continues to line 72, where it checks if the player ran out of guesses. If so, the program tells the player they've lost.

At this point, execution jumps back to the `while` loop on line 60 where it lets the player have another guess. If the player ran out of guesses (or it broke out of the loop with the `break` statement on line 71), then execution would proceed past the loop and to line 75.

Asking the Player to Play Again

```
75.      if not playAgain():
76.          break
```

Line 75 asks the player if they want to play again by calling the `playAgain()` function. If `playAgain()` returns `False`, break out of the `while` loop that started on line 55. Since there's no more code after this loop, the program terminates.

If `playAgain()` returned `True`, then the execution would not execute the `break` statement and execution would jump back to line 55. The program generates a new secret number so the player can play a new game.

Summary

Bagels is a simple game to program but can be difficult to win at. But if you keep playing, you'll eventually discover better ways to guess and make use of the clues the game gives you. This is much like how you'll get better at programming you more you keep at it.

This chapter introduced a few new functions and methods (`random.shuffle()`, `sort()`, and `join()`), along with a couple handy shortcuts. An augmented assignment operators involve less typing when you want to change a variable's relative value such as in `spam = spam + 1`, which can be shortened to `spam += 1`. String interpolation can make your code much more readable by placing `%s` (called a conversion specifier) inside the string instead of using many string concatenation operations.

The next chapter isn't about programming directly, but will be necessary for the games we want to create in the later chapters of this book. We will learn about the math concepts of Cartesian coordinates and negative numbers. These are used in the Sonar, Reversi, and Dodger games, but Cartesian coordinates and negative numbers are used in many games. If you already know about these concepts, give the next chapter a brief read anyway to refresh yourself.

Chapter 12

CARTESIAN COORDINATES

> Topics Covered In This Chapter:
> - Cartesian coordinate systems
> - The X-axis and Y-axis
> - The Commutative Property of Addition
> - Absolute values and the `abs()` function

This chapter doesn't introduce a new game. Instead it goes over some simple mathematical concepts you will use in the rest of this book. In 2D games the graphics on the screen can move left or right and up or down. These two directions make up two-dimensional, or 2D, space. Games with objects moving around a two-dimensional computer screen need a way to translate a place on the screen into integers the program can deal with.

This is where the Cartesian coordinate system comes in. The coordinates are numbers for a specific point on the screen. These numbers can be stored as integers in your program's variables.

Grids and Cartesian Coordinates
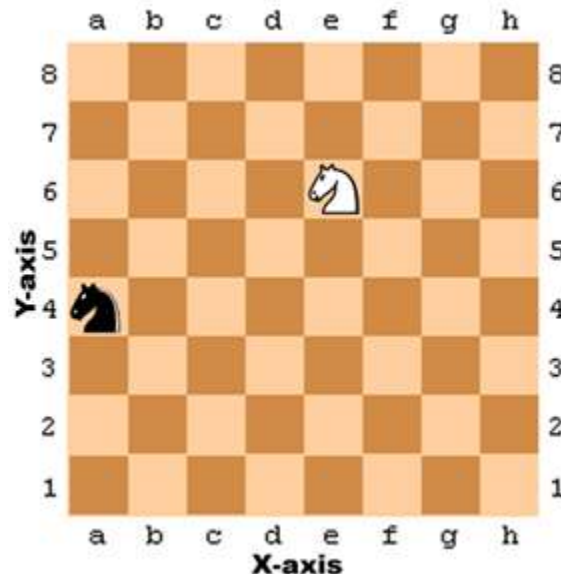
Figure 12-1: A sample chessboard with a black knight at a, 4 and a white knight at e, 6.

A common way to refer to specific places on a chessboard is by marking each row and column with letters and numbers. Figure 12-1 is a chessboard that has each row and each column marked.

A coordinate for a space on the chessboard is a combination of a row and a column. In chess, the knight piece looks like a horse head. The white knight in Figure 12-1 is located at the point e, 6 and the black knight is located at point a, 4.

This labeled chessboard is a Cartesian coordinate system. By using a row label and column label, you can give a coordinate that is for one and only one space on the board. If you've learned about Cartesian coordinate systems in math class, you may know that numbers are used for both the rows and columns. That chessboard would look like Figure 12-2.
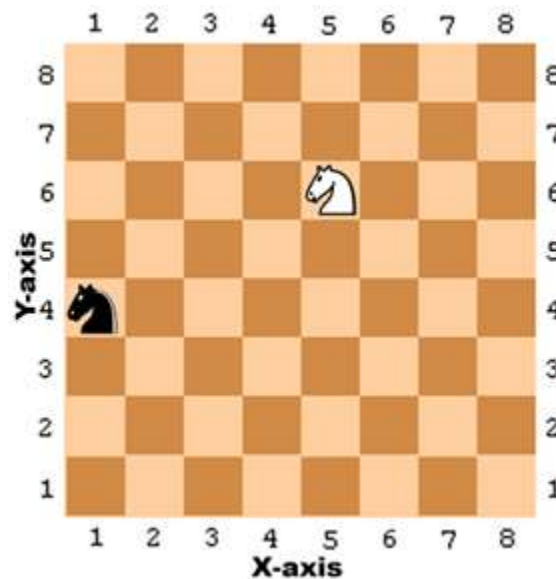
Figure 12-2: The same chessboard but with numeric coordinates for both rows and columns.

The numbers going left and right along the columns are part of the **X-axis**. The numbers going up and down along the rows are part of the **Y-axis**. Coordinates are always described with the X-coordinate first, followed by the Y-coordinate. In Figure 12-2, the white knight is located at the coordinate 5, 6 and not 6, 5. The black knight is located at the coordinate 1, 4 which is not to be confused with 4, 1.

Notice that for the black knight to move to the white knight's position, the black knight must move up two spaces and to the right by four spaces. But you don't need to look at the board to figure this out. If you know the white knight is located at 5, 6 and the black knight is located at 1, 4, then you can use subtraction to figure out this information.

Subtract the black knight's X-coordinate and white knight's X-coordinate: 5 - 1 = 4. The black knight has to move along the X-axis by four spaces.

Subtract the black knight's Y-coordinate and white knight's Y-coordinate: 6 - 4 = 2. The black knight has to move along the Y-axis by two spaces.

By doing some math with the coordinate numbers, you can figure out the distances between two coordinates.

Negative Numbers

Cartesian coordinates use negative numbers. **Negative numbers** are numbers that are smaller than zero. A minus sign in front of a number shows it is negative. -1 is smaller than 0. And -2 is smaller than -1. If you think of regular numbers (called **positive numbers**) as starting from 1 and increasing, you can think of negative numbers as starting from -1 and decreasing. 0 itself isn't positive or negative. In Figure 12-3, you can see the positive numbers increasing to the right and the negative numbers decreasing to the left on a number line.
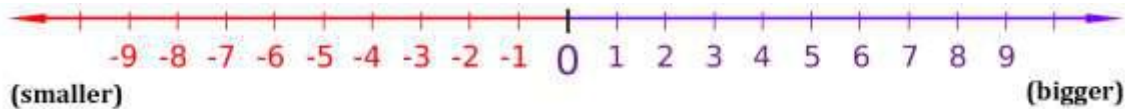
Figure 12-3: A number line.

The number line is useful to see subtraction and addition done with negative numbers. The expression 4 + 3 can be thought of as the white knight starting at position 4 and moving 3 spaces over to the right (addition means increasing, which is in the right direction).
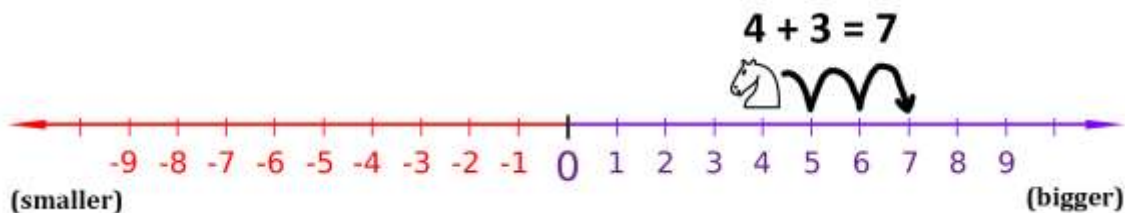
Figure 12-4: Moving the white knight to the right adds to the coordinate.

As you can see in Figure 12-4, the white knight ends up at position 7. This makes sense, because 4 + 3 is 7.

Subtraction is done by moving the white knight to the left. Subtraction means decreasing, which is in the left direction. 4 - 6 would be the white knight starting at position 4 and moving 6 spaces to the left, like in Figure 12-5.
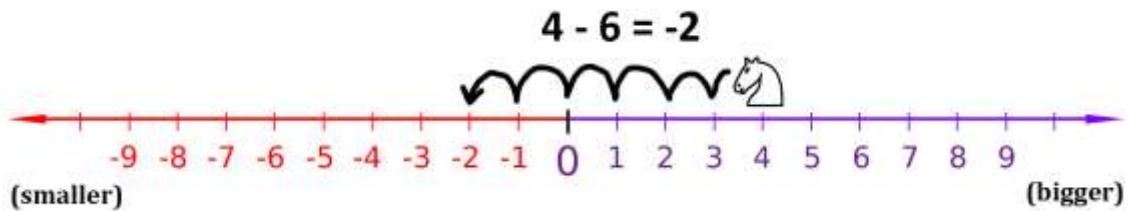
Figure 12-5: Moving the white knight to the left subtracts from the coordinate.

The white knight ends up at position -2. That means 4 - 6 equals -2.

If you add or subtract a negative number, the white knight would move in the *opposite* direction. If you add a negative number, the knight moves to the *left*. If you subtract a negative number, the knight moves to the *right*. The expression -6 - -4 would be equal to -2. The knight starts at -6 and moves to the *right* by 4 spaces. Notice that -6 - -4 has the same answer as -6 + 4.
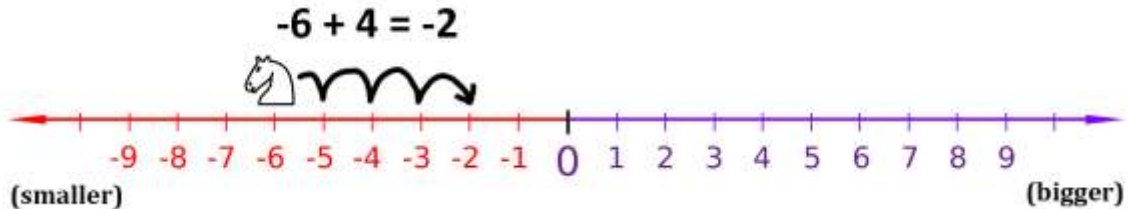
Figure 12-6: Even if the white knight starts at a negative coordinate, moving right still adds to the coordinate.
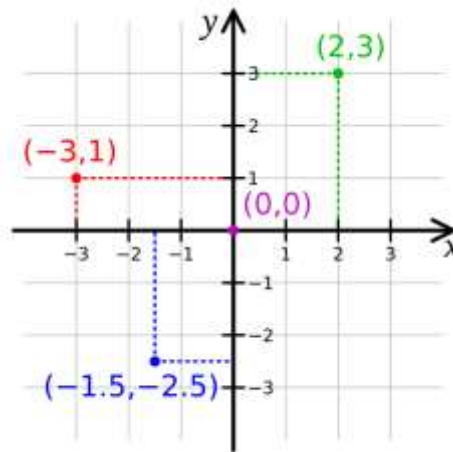
Figure 12-7: Putting two number lines together creates a Cartesian coordinate system.

You can think of the X-axis as a number line. Add another number line going up and down for the Y-axis. If you put these two number lines together, you have a Cartesian coordinate system like in Figure 12-7.

Adding a positive number (or subtracting a negative number) would move the knight up the number line, and subtracting a positive number (or adding a negative number) would move the knight down.

The 0, 0 coordinate is called the **origin**.

Math Tricks

Subtracting and adding negative numbers is easy when you have a number line in front of you. It can also be easy without a number line too. Here are three tricks to help you add and subtract negative numbers by yourself.

Trick 1: "A Minus Eats the Plus Sign on its Left"

When you see a minus sign with a plus sign on the left, you can replace the plus sign with a minus sign. Imagine the minus sign "eating" the plus sign to its left. The answer is still the same, because adding a negative value is the same as subtracting a positive value. 4 + -2 and 4 - 2 both evaluate to 2.

$$4 + -2 = 2$$

(a minus eats the plus sign on its left)

$$4 - 2 = 2$$

Figure 12-8: Trick 1 - Adding a positive and negative number.

Trick 2: "Two Minuses Combine Into a Plus"

When you see the two minus signs next to each other without a number between them, they can combine into a plus sign. The answer is still the same, because subtracting a negative value is the same as adding a positive value.

$$4 - -2 = 6$$

(two minuses combine into a plus)

$$4 + 2 = 6$$

Figure 12-9: Trick 2 - Subtracting a positive and negative number.

Trick 3: The Commutative Property of Addition

You can always swap the numbers in addition. This is the **commutative property** of addition. That means that doing a swap like $6 + 4$ to $4 + 6$ will not change the answer.

If you count the boxes in Figure 12-10, you can see that it doesn't matter if you swap the numbers for addition.

$$6+4=10$$

$$4+6=10$$

Figure 12-10: Trick 3 - The commutative property of addition.

Say you are adding a negative number and a positive number, like $-6 + 8$. Because you are adding numbers, you can swap the order of the numbers without changing the answer. $-6 + 8$ is the same as $8 + -6$.

Then when you look at $8 + -6$, you see that the minus sign can eat the plus sign to its left, and the problem becomes $8 - 6 = 2$. But this means that $-6 + 8$ is also 2! You've rearranged the problem to have the same answer, but made it easier for us to solve without using a calculator or computer.

$$-6 + 8 = 2$$

(because this is addition, swap the order)

$$8 + -6 = 2$$

(the minus sign eats the plus sign on its left)

$$8 - 6 = 2$$

Figure 12-11: Using the math tricks together.

Absolute Values and the abs() Function

The **absolute value** of a number is the number without the negative sign in front of it. Therefore, positive numbers do not change, but negative numbers become positive. For example, the absolute value of -4 is 4. The absolute value of -7 is 7. The absolute value of 5 (which is positive) is just 5.

You can figure out the distance between two objects by subtracting their positions and taking the absolute value of the difference. Imagine that the white knight is at position 4 and the black knight is at position -2. The distance would be 6, since 4 - -2 is 6, and the absolute value of 6 is 6.

It works no matter what the order of the numbers is. -2 - 4 (that is, negative two minus four) is -6, and the absolute value of -6 is also 6.

Python's abs() function returns the absolute value of an integer. Try entering the following into the interactive shell:

```
>>> abs(-5)
5
>>> abs(42)
42
>>> abs(-10.5)
10.5
```

Coordinate System of a Computer Screen
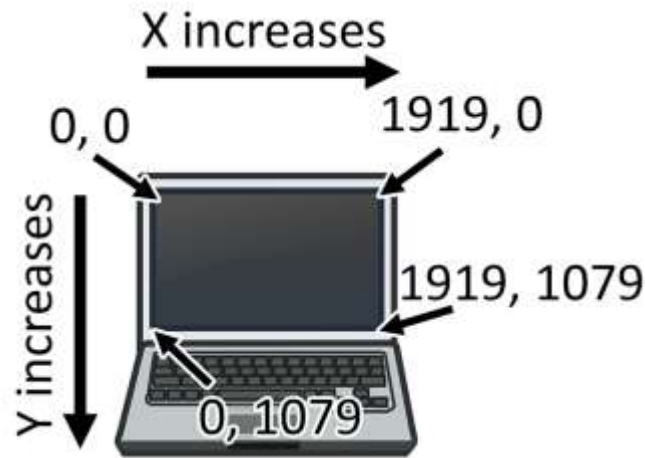
Figure 12-12: The Cartesian coordinate system on a computer screen.

It is common that computer screens use a coordinate system that has the origin (0, 0) at the top left corner of the screen, which increases going down and to the right. This is shown in Figure 12-12. There are no negative coordinates. Most computer graphics use this coordinate system, and you will use it in this book's games.

Summary

Most programming doesn't require understanding a lot of math. Up until this chapter, we've been getting by on simple addition and multiplication.

Cartesian coordinate systems are needed to describe where in a two-dimensional area a certain position is. Coordinates have two numbers: the X-coordinate and the Y-coordinate. The X-axis runs left and right and the Y-axis runs up and down. On a computer screen, origin is in the top-left corner and the coordinates increase going right and down.

The three tricks you learned in this chapter make it easy to add positive and negative integers. The first trick is that a minus sign will eat the plus sign on its left. The second trick is that two minuses next to each other will combine into a plus sign. The third trick is that you can swap the position of the numbers you are adding.

For the rest of the book, we will use the concepts from this chapter in our games because they have two-dimensional areas in them. All graphical games require understanding how Cartesian coordinates work.

Chapter 13

SONAR TREASURE HUNT

Topics Covered In This Chapter:
- Data structures
- The remove() list method
- The isdigit() string method
- The sys.exit() function

The game in this chapter is the first to make use of Cartesian coordinates that you learned about in Chapter 12. The game also has **data structures** (which is just a fancy way of saying complex variables such as those that contain lists of lists.) As the games you program become more complicated, you'll need to organize your data in data structures.

In this chapter's game, the player places sonar devices at various places in the ocean to locate sunken treasure chests. Sonar is a technology that ships use to locate objects under the sea. The sonar devices (in this game) will tell the player how far away the closest treasure chest is, but not in what direction. But by placing multiple sonar devices down, the player can figure out where the treasure chest is.

There are three chests to collect, but the player has only sixteen sonar devices to use to find them. Imagine that you could not see the treasure chest in the following picture. Because each sonar device can only find the distance, not direction, the possible places the treasure could be is anywhere in a square ring around the sonar device (see Figure 13-1).
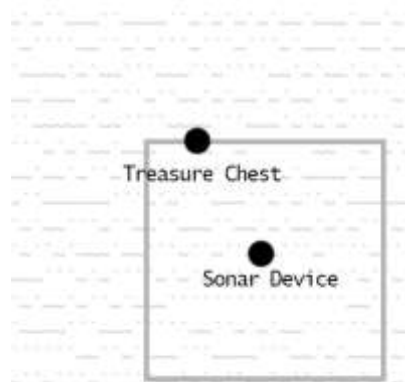
Figure 13-1: The sonar device's square ring touches the (hidden) treasure chest.

Figure 13-2: Combining multiple square rings of shows where treasure chests could be.

But multiple sonar devices working together can narrow it to an exact place where the rings intersect each other. See Figure 13-2. (Normally these rings would be circles, but this game will use squares to make programming it easier.)

Sample Run of Sonar Treasure Hunt

```
S O N A R !
Would you like to view the instructions? (yes/no)
no
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890123456789
 0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~`~`~`~~```~`~```~`~~~~`` 0
 1 ~`~~~``~~~~`~`~~`~`~`~~``~`~`~~`~`~~~~~`~````~`~~`~~~~` 1
 2 `~``~`~~~`~`~`~`~`~`````~~~~~~~~~`~`~~`~``~~~~~```~~`~`` 2
 3 ``~`~~`~`~`~`~`~`~~`~`~~`~``~~~`~`~`````~`````~~~~``~`` 3
 4 ``~~`~~~`~`~~````~`~`~`~`~~~`~~``~`~~`~~`~`~`~~`~~~~``` 4
 5 ~~```~~~`~`~~`~`~`~`~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~~~` 5
 6 ``~~````~~~~`~`~~~``~~~~`~`~~`~~`~~`~~~`~~~`~`~~~``~~~ 6
 7 `~```````~`~``~`~~`~~~`~~`~`~~~``~`~~`~`~`~~``~~ 7
 8 `~```~``~`~~`~~~`~~`~~~``~`~`~~~``~`~`~``~~```~`~~~~` 8
 9 ~``~~`~`~`~`~~`~`~`~~``~`~`~~~~`~`~`~~~`~`~`~~~``~~``` 9
10 ```~`~`~`~`~`~`~~`~`~```~`~~~`~~`~~`~~~~`~~~`~`~~```~ 10
11 ``~``~~~`~``~~`~~~`~```~`~~`~`~~`~~`~`~~`~~~```~```~` 11
12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~`~```~~~```~~`~~`~``~`~`~ 12
13 `~~```~~``~``~~~`~```~`~~~~~~~~~`~~``~~~~~````~`~`~`~`~~~~ 13
14 `~~`~`~``~``~`~`~```~~`~~~~`~```~`~`~`~`~~~```~~`````~ 14
    012345678901234567890123456789012345678901234567890123456789
              1         2         3         4         5
You have 16 sonar devices left. 3 treasure chests remaining.
Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)
10 10
```

```
              1         2         3         4         5
     012345678901234567890123456789012345678901234567890123456789
  0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~`~`~~```~`~```~`~~~~``  0
  1 ~`~~~``````~~~~`~`~~`~``~`~~``~`~`~~`~`~~~~~~`~`````~`~~`~~~~`  1
  2 `~``~`~~~`~`~`~`~`~``````~~~~~~~~~`~`~~`~``~~~~~```~~`~```  2
  3 ``~`~~``~`~`~``~`~~`~~`~`~```~`~``````~`~``````~~~~``~``  3
  4 ``~~`~~~`~`~~```~`~`~`~```~~~`~~```~`~`~~`~~`~`~`~~`~~~~```  4
  5 ~~```~~~`~`~~``~`~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~~~~`  5
  6 ``~~```~`~~~~`~`~~~```~~~~`~~`~~`~~```~~`~~~`~~~`~`~~~``~~~  6
  7 `~````````~`~`~``~`~`~~`~~~~`~~`~``~`~~~```~`~~`~``~~```~~  7
  8 `~````~``~`~~`~~~`~~``~~~`~```~`~``~`~``~``~`~~```~`~~~~~`  8
  9 ~```~~`~`~``~`~~``~`~``~``~~~~`~~`~~~`~`~`~~~`~`~~``  9
 10 ```~`~```~5`~`~`~~`~`~````~`~~~`~~`~~`~~~~`~~~`~`~~```~ 10
 11 ```~``~~~`~`~``~~`~~~`~```~`~~`~`~~`~~`~`~~`~~~```~`````~ 11
 12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~``~```~~~``~~`~~`~`~`~`~`~ 12
 13 `~~```~~``~`~``~~~`~```~`~~~~~~~~~`~~``~~~~~`````~`~`~`~~~~ 13
 14 `~~`~`~`````~```~`~`~```~~`~~~~`~``~`~`~`~``~~~```~~`````~ 14
     012345678901234567890123456789012345678901234567890123456789
              1         2         3         4         5
Treasure detected at a distance of 5 from the sonar device.
You have 15 sonar devices left. 3 treasure chests remaining.
Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)
15 6
              1         2         3         4         5
     012345678901234567890123456789012345678901234567890123456789
  0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~`~`~~```~`~```~`~~~~``  0
  1 ~`~~~``````~~~~`~`~~`~``~`~~``~`~`~~`~`~~~~~~`~```~`~~`~~~~`  1
  2 `~``~`~~~`~`~`~`~`~``````~~~~~~~~~`~`~~`~``~~~~~```~~`~```  2
  3 ``~`~~``~`~`~``~`~~`~~`~`~```~`~``````~`~```````~~~~``~``  3
  4 ``~~`~~~`~`~~```~`~`~`~```~~~`~~```~`~`~~`~~`~`~`~~`~~~~```  4
  5 ~~```~~~`~`~~``~`~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~~~~`  5
  6 ``~~````~~~~`~4~~~``~~~~`~~`~~`~~```~~`~~~`~~~`~`~~~``~~~  6
  7 `~````````~`~`~``~`~`~~`~~~~`~~`~``~`~~~```~`~~`~``~~```~~  7
  8 `~````~``~`~~`~~~`~~``~~~`~```~`~``~`~``~``~`~~```~`~~~~~`  8
  9 ~```~~`~`~``~`~~``~`~``~``~~~~`~~`~~~`~`~`~~~`~`~~``  9
 10 ```~`~```~5`~`~`~~`~`~````~`~~~`~~`~~`~~~~`~~~`~`~~```~ 10
 11 ```~``~~~`~`~``~~`~~~`~```~`~~`~`~~`~~`~`~~`~~~```~`````~ 11
 12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~``~```~~~``~~`~~`~`~`~`~`~ 12
 13 `~~```~~``~`~``~~~`~```~`~~~~~~~~~`~~``~~~~~`````~`~`~`~~~~ 13
 14 `~~`~`~`````~```~`~`~```~~`~~~~`~``~`~`~`~``~~~```~~`````~ 14
     012345678901234567890123456789012345678901234567890123456789
              1         2         3         4         5
Treasure detected at a distance of 4 from the sonar device.
You have 14 sonar devices left. 3 treasure chests remaining.
Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)
15 10
              1         2         3         4         5
```

```
      012345678901234567890123456789012345678901234567890123456789
 0  `~~~`~~~`~`~~`~~~~~`~`` ~~~~`~`~~~`~``~``~~````~``~``~`~~~~`` 0
 1  ~`~~~```~~~~`~`~~`~``~`~~```~`~`~~`~`~~~~~~`~````~`~~`~~~~` 1
 2  `~``~`~~~`~`~`~`~`~`````~~~~~~~~~`~`~~`~``~~~~~```~~`~``` 2
 3  ``~`~~`~`~`~`~`~`~~`~`~`~`~`~~`~`~`````~``````~~~~``~` 3
 4  ``~~`~~~`~`~~`~`~`~`~`~`~~~`~`~~`~`~`~``~~~~`` 4
 5  ~~```~~~`~`~~`~`~`~`````~`~~`~~~~~`~~`~`~`~~~`~~`~`~`~`~~~` 5
 6  ``~~````~~~~`~0~~~``~~~~`~~`~~`~~``~~`~~~`~~~`~`~~~`~~~ 6
 7  `~````````~`~``~`~``~~`~~~~`~~`~`~~~```~`~~~`~`~`~~```~~ 7
 8  `~```~`~`~~`~~~`~~`~~~`~`~`~~~`~``~`~~``~``~`~~~~` 8
 9  ~`~``~~`~`~``~`~~`~`~``~`~`~`~~~~`~`~~~`~`~`~~~``~~`` 9
10  ```~`~```~0`~``0`~~`~``~``~``~~~`~~`~~``~~~~`~~~`~`~~```~ 10
11  ```~```~~~`~```~~`~~~`~```` ~`~~`~`~~`~~`~`~~`~~~````~```~`~`` 11
12  ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~```~~~```~~`~~`~`~`~`~ 12
13  `~~```~~`~`~```~~~`~``~`~~~~~~~~~~`~~`~~~~~````~`~`~`~~~~ 13
14  `~~`~`~```~```~`~`~```~~`~~~~`~`~``~`~`~``~~~```~~`````~ 14
      012345678901234567890123456789012345678901234567890123456789
                   1         2         3         4         5
You have found a sunken treasure chest!
You have 13 sonar devices left. 2 treasure chests remaining.
Where do you want to drop the next sonar device? (0-59 0-14) (or type quit)


...skipped over for brevity...


                   1         2         3         4         5
      012345678901234567890123456789012345678901234567890123456789
 0  `~~~`~~~`~`~~`~~~~~`~`` ~~~~`~`~~~`~``~``~~````~``~``~`~~~~`` 0
 1  ~`~~~```~~~~`~`~~`~``~`~~```~0~`~~`~`~~~~~~`~````~`~~`~~~~` 1
 2  `~``~`~~~`~`~`~`~`~`````~~0~~~0~~`~`~~`~``~~~~~```~~`~``` 2
 3  ``~3~~``8`~`~`~`~`~~`~`~`~`~`~~`~`0~````~`````~~~~``~` 3
 4  ``~~`~~~`~`~~`~`~``~`~`~`~0`~~0``~~``~`~~`~~`~`~`~~`~~~~`` 4
 5  ~~```~~~`~`~~`~`~`~`````~`~~`~~~~~`~~`~`~`~~~`~~`~`~`~`~~~` 5
 6  ``~~````~~~~`~0~~~``~~~~`~~`~~`~~``~~`~~~`~~~`0`~~~``~~~ 6
 7  `~````````~`~``~`~``~~`~~~~`~~`~`~~~```~`~~~`~`~`~~```~~ 7
 8  `~```~`~`~`~~`~~~`~~`~~~`~`~`~~~`~`0```0`~`~~```~`~~~~` 8
 9  ~```~~`~`~``~`~~`~`~``~```~0~``~~~~`~`~`~~~`~`~`~~~``~~`` 9
10  ```~`~```~0`~``0`~~`~``~``~``~~~`~~`~~``~~~~`~~~`~`~~```~ 10
11  ```~```~~~`~```~~`~~~`~````` ~`~~`~`~~`~~`~`~~`~~~````~```~` 11
12  ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~```~~~```~~`~~`~`~`~`~ 12
13  `~~```~~`~`~```~~~`~``~`~~~~~~~~~~`~~`~~~~~````~`~`~`~~~~ 13
14  `~~`~`~```~```~`~`~```~~`~~~~`~`~``~`~`~``~~~```~~`````~ 14
      012345678901234567890123456789012345678901234567890123456789
                   1         2         3         4         5
Treasure detected at a distance of 4 from the sonar device.
We've run out of sonar devices! Now we have to turn the ship around and head
for home with treasure chests still out there! Game over.
    The remaining chests were here:
```

```
    0, 4
Do you want to play again? (yes or no)
no
```

Source Code of Sonar Treasure Hunt

Below is the source code for the game. Type it into a new file, then save the file as *sonar.py* and run it by pressing the **F5** key. If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/sonar.

```
                                                                    sonar.py
  1. # Sonar
  2.
  3. import random
  4. import sys
  5.
  6. def drawBoard(board):
  7.     # Draw the board data structure.
  8.
  9.     hline = '    ' # initial space for the numbers down the left side of
the board
 10.     for i in range(1, 6):
 11.         hline += (' ' * 9) + str(i)
 12.
 13.     # print the numbers across the top
 14.     print(hline)
 15.     print('  ' + ('0123456789' * 6))
 16.     print()
 17.
 18.     # print each of the 15 rows
 19.     for i in range(15):
 20.         # single-digit numbers need to be padded with an extra space
 21.         if i < 10:
 22.             extraSpace = ' '
 23.         else:
 24.             extraSpace = ''
 25.         print('%s%s %s %s' % (extraSpace, i, getRow(board, i), i))
 26.
 27.     # print the numbers across the bottom
 28.     print()
 29.     print('  ' + ('0123456789' * 6))
 30.     print(hline)
 31.
 32.
 33. def getRow(board, row):
```

```
34.      # Return a string from the board data structure at a certain row.
35.      boardRow = ''
36.      for i in range(60):
37.          boardRow += board[i][row]
38.      return boardRow
39.
40.  def getNewBoard():
41.      # Create a new 60x15 board data structure.
42.      board = []
43.      for x in range(60): # the main list is a list of 60 lists
44.          board.append([])
45.          for y in range(15): # each list in the main list has 15 single-
character strings
46.              # use different characters for the ocean to make it more
readable.
47.              if random.randint(0, 1) == 0:
48.                  board[x].append('~')
49.              else:
50.                  board[x].append('`')
51.      return board
52.
53.  def getRandomChests(numChests):
54.      # Create a list of chest data structures (two-item lists of x, y int
coordinates)
55.      chests = []
56.      for i in range(numChests):
57.          chests.append([random.randint(0, 59), random.randint(0, 14)])
58.      return chests
59.
60.  def isValidMove(x, y):
61.      # Return True if the coordinates are on the board, otherwise False.
62.      return x >= 0 and x <= 59 and y >= 0 and y <= 14
63.
64.  def makeMove(board, chests, x, y):
65.      # Change the board data structure with a sonar device character.
Remove treasure chests
66.      # from the chests list as they are found. Return False if this is an
invalid move.
67.      # Otherwise, return the string of the result of this move.
68.      if not isValidMove(x, y):
69.          return False
70.
71.      smallestDistance = 100 # any chest will be closer than 100.
72.      for cx, cy in chests:
73.          if abs(cx - x) > abs(cy - y):
74.              distance = abs(cx - x)
75.          else:
```

```
 76.                  distance = abs(cy - y)
 77.
 78.          if distance < smallestDistance: # we want the closest treasure
chest.
 79.                  smallestDistance = distance
 80.
 81.      if smallestDistance == 0:
 82.          # xy is directly on a treasure chest!
 83.          chests.remove([x, y])
 84.          return 'You have found a sunken treasure chest!'
 85.      else:
 86.          if smallestDistance < 10:
 87.              board[x][y] = str(smallestDistance)
 88.              return 'Treasure detected at a distance of %s from the sonar
device.' % (smallestDistance)
 89.          else:
 90.              board[x][y] = 'O'
 91.              return 'Sonar did not detect anything. All treasure chests out
of range.'
 92.
 93.
 94. def enterPlayerMove():
 95.      # Let the player type in their move. Return a two-item list of int xy
coordinates.
 96.      print('Where do you want to drop the next sonar device? (0-59 0-14)
(or type quit)')
 97.      while True:
 98.          move = input()
 99.          if move.lower() == 'quit':
100.              print('Thanks for playing!')
101.              sys.exit()
102.
103.          move = move.split()
104.          if len(move) == 2 and move[0].isdigit() and move[1].isdigit() and
isValidMove(int(move[0]), int(move[1])):
105.              return [int(move[0]), int(move[1])]
106.          print('Enter a number from 0 to 59, a space, then a number from 0
to 14.')
107.
108.
109. def playAgain():
110.      # This function returns True if the player wants to play again,
otherwise it returns False.
111.      print('Do you want to play again? (yes or no)')
112.      return input().lower().startswith('y')
113.
114.
```

```
115. def showInstructions():
116.     print('''Instructions:
117. You are the captain of the Simon, a treasure-hunting ship. Your current
mission
118. is to find the three sunken treasure chests that are lurking in the part
of the
119. ocean you are in and collect them.
120.
121. To play, enter the coordinates of the point in the ocean you wish to drop
a
122. sonar device. The sonar can find out how far away the closest chest is to
it.
123. For example, the d below marks where the device was dropped, and the 2's
124. represent distances of 2 away from the device. The 4's represent
125. distances of 4 away from the device.
126.
127.     444444444
128.     4       4
129.     4 22222 4
130.     4 2   2 4
131.     4 2 d 2 4
132.     4 2   2 4
133.     4 22222 4
134.     4       4
135.     444444444
136. Press enter to continue...''')
137.     input()
138.
139.     print('''For example, here is a treasure chest (the c) located a
distance of 2 away
140. from the sonar device (the d):
141.
142.     22222
143.     c   2
144.     2 d 2
145.     2   2
146.     22222
147.
148. The point where the device was dropped will be marked with a 2.
149.
150. The treasure chests don't move around. Sonar devices can detect treasure
151. chests up to a distance of 9. If all chests are out of range, the point
152. will be marked with 0
153.
154. If a device is directly dropped on a treasure chest, you have discovered
155. the location of the chest, and it will be collected. The sonar device will
156. remain there.
```

```
157.
158. When you collect a chest, all sonar devices will update to locate the next
159. closest sunken treasure chest.
160. Press enter to continue...''')
161.     input()
162.     print()
163.
164.
165. print('S O N A R !')
166. print()
167. print('Would you like to view the instructions? (yes/no)')
168. if input().lower().startswith('y'):
169.     showInstructions()
170.
171. while True:
172.     # game setup
173.     sonarDevices = 16
174.     theBoard = getNewBoard()
175.     theChests = getRandomChests(3)
176.     drawBoard(theBoard)
177.     previousMoves = []
178.
179.     while sonarDevices > 0:
180.         # Start of a turn:
181.
182.         # show sonar device/chest status
183.         if sonarDevices > 1: extraSsonar = 's'
184.         else: extraSsonar = ''
185.         if len(theChests) > 1: extraSchest = 's'
186.         else: extraSchest = ''
187.         print('You have %s sonar device%s left. %s treasure chest%s
remaining.' % (sonarDevices, extraSsonar, len(theChests), extraSchest))
188.
189.         x, y = enterPlayerMove()
190.         previousMoves.append([x, y]) # we must track all moves so that
sonar devices can be updated.
191.
192.         moveResult = makeMove(theBoard, theChests, x, y)
193.         if moveResult == False:
194.             continue
195.         else:
196.             if moveResult == 'You have found a sunken treasure chest!':
197.                 # update all the sonar devices currently on the map.
198.                 for x, y in previousMoves:
199.                     makeMove(theBoard, theChests, x, y)
200.             drawBoard(theBoard)
201.             print(moveResult)
```

```
202.
203.         if len(theChests) == 0:
204.             print('You have found all the sunken treasure chests!
Congratulations and good game!')
205.             break
206.
207.         sonarDevices -= 1
208.
209.     if sonarDevices == 0:
210.         print('We\'ve run out of sonar devices! Now we have to turn the
ship around and head')
211.         print('for home with treasure chests still out there! Game over.')
212.         print('    The remaining chests were here:')
213.         for x, y in theChests:
214.             print('    %s, %s' % (x, y))
215.
216.     if not playAgain():
217.         sys.exit()
```

Designing the Program

Before trying to understand the source code, play the game a few times first to understand what is going on. The Sonar game uses lists of lists and other such complicated variables, called **data structures**. Data structures are variables that store arrangements of values to represent something. For example, in the Tic Tac Toe chapter, a Tic Tac Toe board data structure was a list of strings. The string represented an X, O, or empty space and the index of the string in the list represented the space on the board. The Sonar game will have similar data structures for the locations of treasure chests and sonar devices.

How the Code Works

```
1. # Sonar
2.
3. import random
4. import sys
```

Lines 3 and 4 import modules `random` and `sys`. The `sys` module contains the `exit()` function, which causes the program to terminate immediately. This function is used later in the program.

Drawing the Game Board

```
6. def drawBoard(board):
```

The Sonar game's board is an ASCII art ocean with X- and Y-axis coordinates around it. The back tick (`) and tilde (~) characters are located next to the 1 key on your keyboard will be used for the ocean waves. It looks like this:

```
             1         2         3         4         5
    012345678901234567890123456789012345678901234567890123456789
 0 ~~~`~``~~~``~~~~``~`~`~`~`~~`~~~`~~`~`````~~`~``~`~~```~`~` 0
 1 `~`~```~~`~`~```~```~``~`~~~`~~`~~~`````~`~``~~``~~`~~` 1
 2 ```~~~~`~`~~```~~~``~```~~`~`~~`~`~`~``~~`~``~~`~`~~~~~`~ 2
 3 ~~~~`~~~``~```~``~~`~`~~``~```~`~```~``~`~`~`~```~ 3
 4 ~```~~~~~`~~```~~~~```~~~`~`~`~``~`~~`~`~~``~~`~``~`~`~~ 4
 5 `~```~`~`~~`~~~```~~`~`````~~``~`~`~~~~`~~``~~~~~~`~``~~` 5
 6 ``~~`~~`~`~`~```~`~```~~``~`~~~~`~~```~~~`~`~`~~`~~~```~~~ 6
 7 ``~`~`~~~~~~```~`~```~~~`~`~`~`~~~~~~````~~~`~~`~~`~~`~~ 7
 8 ~~`~`~~```~`~~`~~~``~~`~`~~`~`~```~```~~~```~~~~~~`~`~~~~` 8
 9 ```~``~`~~~`~~```~`~``~~~```~``~```~`~~`~~~~~`~``~~~~~``` 9
10 `~~~~``~`~``~```~`~~`~`~~~~`~``~``~```~~``````~`~``~``````` 10
11 ~~`~`~~`~``~`~~~```````````````````~~````~`~~``~`~~~~`~~~`~~``~ 11
12 ~~`~~~~```~~~```~``~~``~`~`~~`````~``~~`~`````~~`~~~`~~`~ 12
13 `~`````~`~~``~`~~~```~~~~```~~`~`~~~`~``````~~`~``~`~`~~~~ 13
14 ~~~``~```~````~~`~`~`~~`~``~`~~`~`~``~`~``~~``~`~``~```~~~ 14
    012345678901234567890123456789012345678901234567890123456789
             1         2         3         4         5
```

The drawing in the `drawBoard()` function has four steps.

- First, create a string variable of the line with 1, 2, 3, 4, and 5 spaced out with wide gaps (to mark the coordinates for 10, 20, 30, 40, and 50 on the X-axis).

- Second, use that string to display the X-axis coordinates along the top of the screen.

- Third, print each row of the ocean along with the Y-axis coordinates on both sides of the screen.

- Fourth, print the X-axis again at the bottom. Coordinates on all sides makes it easier to see coordinates for where to place a sonar device.

Drawing the X-Coordinates Along the Top

```
  7.      # Draw the board data structure.
  8.
  9.      hline = '    ' # initial space for the numbers down the left side of
the board
 10.      for i in range(1, 6):
 11.          hline += (' ' * 9) + str(i)
```

Look again at the top part of the board in Figure 13-3. It has + plus signs instead of blank spaces so you can count the blank spaces easier:

```
+++++++++++++1+++++++++2+++++++++3 # first line
+++012345678901234567890123456789 # second line

+0 ~~~`~``~~~``~~~~``~`~`~`~`~~`~~~~```~`~` 0 # third line
```

Figure 13-3: The spacing used for printing the top of the game board.

The numbers on the first line which mark the tens position all have nine spaces between them, and there are thirteen spaces in front of the 1. Lines 9 to 11 create this string with this line and store it in a variable named `hline`.

```
13.      # print the numbers across the top
14.      print(hline)
15.      print('   ' + ('0123456789' * 6))
16.      print()
```

To print the numbers across the top of the sonar board, first print the contents of the `hline` variable. Then on the next line, print three spaces (so that this row lines up correctly), and then print the string `'012345678901234567890123456789012345678901234567890123456789'`. But as a shortcut you can use (`'0123456789' * 6`), which evaluates to the same string.

Drawing the Rows of the Ocean

```
18.      # print each of the 15 rows
19.      for i in range(15):
20.          # single-digit numbers need to be padded with an extra space
21.          if i < 10:
22.              extraSpace = ' '
23.          else:
24.              extraSpace = ''
25.          print('%s%s %s %s' % (extraSpace, i, getRow(board, i), i))
```

Lines 19 to 25 print each row of ocean waves, including the numbers down the side to label the Y-axis. The `for` loop prints rows 0 through 14, along with the row numbers on either side of the board.

There's a small problem. Numbers with only one digit (like 0, 1, 2, and so on) only take up one space when printed, but numbers with two digits (like 10, 11, and 12) take up two spaces. The rows won't line up if the coordinates have different sizes. It will look like this:

```
8  ~~`~`~~```~``~~``~~~``~~`~`~~`~`~``~```~`````~~~```~~~~~`~`~~~~` 8
9  ```~``~`~~~`~~```~``~``~~~```~````~```~`~~`~~~~~`~`~~~~~``` 9
10 `~~~~```~`~```~`~`~~`~`~~~~`~``~`~``~`~```~~``````~`~``~`````` 10
11 ~~`~`~~`~`~`~~~``````````````````~~````~`~~``~`~~~~`~~~`~~`~ 11
```

The solution is easy. Add a space only in front of all the single-digit numbers. Lines 21 to 24 set the variable extraSpace to either a space or an empty string. The extraSpace variable is always printed, but only has a space character in it for single-digit row numbers. Otherwise, it is the empty string. This way, all of the rows will line up when you print them.

The getRow() function takes a row number and returns a string representing that row's ocean waves. Its two parameters are the board data structure stored in the board variable and a row number. Let's look at this function next.

Drawing the X-Coordinates Along the Bottom

```
27.      # print the numbers across the bottom
28.      print()
29.      print('   ' + ('0123456789' * 6))
30.      print(hline)
```

Lines 27 to 30 are similar to lines 13 to 16. They print the X-axis coordinates at the bottom of the screen.

Getting the State of a Row in the Ocean

```
33. def getRow(board, row):
34.     # Return a string from the board data structure at a certain row.
35.     boardRow = ''
36.     for i in range(60):
37.         boardRow += board[i][row]
38.     return boardRow
```

While the board parameter is a data structure for the entire ocean's waves, the getRow() function creates a string for a single row.

First set boardRow to the blank string. The Y-axis coordinate is passed as the row parameter. The string is made by concatenating board[0][row], board[1][row], board[2][row], and so on up to board[59][row]. This is because the row contains 60 characters, from index 0 to index 59.

The for loop on line 36 iterates over integers 0 to 59. On each iteration, the next character in the board data structure is copied on to the end of boardRow. By the time the loop is done, boardRow has the complete row's ASCII art waves and is returned.

Creating a New Game Board

```
40. def getNewBoard():
41.     # Create a new 60x15 board data structure.
42.     board = []
43.     for x in range(60): # the main list is a list of 60 lists
44.         board.append([])
```

A new board data structure is needed at the start of each new game. The board data structure is a list of lists of strings. The first list represents the X coordinate. Since the game's board is 60 characters across, this first list needs to contain 60 lists. Create a for loop that will append 60 blank lists to it.

```
45.         for y in range(15): # each list in the main list has 15 single-
character strings
46.             # use different characters for the ocean to make it more
readable.
47.             if random.randint(0, 1) == 0:
48.                 board[x].append('~')
49.             else:
50.                 board[x].append('`')
```

But board is more than just a list of 60 blank lists. Each of the 60 lists represents an X coordinate of the game board. There are 15 rows in the board, so each of these 60 lists must have 15 characters in them. Line 45 is another for loop to add 15 single-character strings that represent the ocean.

The "ocean" will be a bunch of randomly chosen '~' and '`' strings. If the return value of random.randint() is 0, add the '~' string. Otherwise add the '`' string. This will give the ocean a random, choppy look to it.

Remember that the board variable is a list of 60 lists, each list having 15 strings. That means to get the string at coordinate 26, 12, you would access board[26][12], and not board[12][26]. The X coordinate is first, then the Y coordinate.

```
51.     return board
```

Finally, the function returns the value in the board variable.

Creating the Random Treasure Chests

```
53. def getRandomChests(numChests):
```

```
54.      # Create a list of chest data structures (two-item lists of x, y int
coordinates)
55.      chests = []
56.      for i in range(numChests):
57.          chests.append([random.randint(0, 59), random.randint(0, 14)])
58.      return chests
```

The game also randomly decides where the hidden treasure chests are. The treasure chests are represented as a list of lists of two integers. These two integers will be the X and Y coordinates of a single chest.

For example, if the chest data structure was [[2, 2], [2, 4], [10, 0]], then this would mean there are three treasure chests, one at 2, 2, another chest at 2, 4, and a third one at 10, 0.

The numChests parameter tells the function how many treasure chests to generate. Line 56's for loop will iterate numChests number of times, and on each iteration line 57 appends a list of two random integers. The X coordinate can be anywhere from 0 to 59, and the Y coordinate can be from anywhere between 0 and 14. The expression [random.randint(0, 59), random.randint(0, 14)] that is passed to the append method will evaluate to a list value like [2, 2] or [2, 4] or [10, 0]. This list value is appended to chests.

Determining if a Move is Valid

```
60. def isValidMove(x, y):
61.      # Return True if the coordinates are on the board, otherwise False.
62.      return x >= 0 and x <= 59 and y >= 0 and y <= 14
```

When the player types in X and Y coordinates of where they want to drop a sonar device, they may not type invalid coordinates. The X coordinate must be between 0 and 59 and the Y coordinate must be between 0 and 14.

The isValidMove() function uses a simple expression that uses and operators to ensure that each part of the condition is True. If even one part is False, then the entire expression evaluates to False. This function returns this Boolean value.

Placing a Move on the Board

```
64. def makeMove(board, chests, x, y):
65.      # Change the board data structure with a sonar device character.
Remove treasure chests
66.      # from the chests list as they are found. Return False if this is an
invalid move.
67.      # Otherwise, return the string of the result of this move.
```

```
68.        if not isValidMove(x, y):
69.            return False
```

In the Sonar game, the game board is updated to display a number for each sonar device dropped to show how far away the closest treasure chest is. So when the player makes a move by giving the program an X and Y coordinate, the board changes based on the positions of the treasure chests.

The makeMove() function takes four parameters: the game board data structure, the treasure chests data structure, and the X and Y coordinates. Line 69 returns False if the X and Y coordinates if was passed do not exist on the game board. If isValidMove() returns False, then makeMove() will itself return False.

Otherwise, makeMove() will return a string value describing what happened in response to the move:

- If the coordinates land directly on the treasure, makeMove() returns 'You have found a sunken treasure chest!'.

- If the coordinates are within a distance of 9 or less, makeMove() returns 'Treasure detected at a distance of %s from the sonar device.' (where %s is replaced with the integer distance).

- Otherwise, makeMove() will return 'Sonar did not detect anything. All treasure chests out of range.'.

```
71.        smallestDistance = 100 # any chest will be closer than 100.
72.        for cx, cy in chests:
73.            if abs(cx - x) > abs(cy - y):
74.                distance = abs(cx - x)
75.            else:
76.                distance = abs(cy - y)
77.
78.            if distance < smallestDistance: # we want the closest treasure
chest.
79.                smallestDistance = distance
```

Given the coordinates of where the player wants to drop the sonar device and a list of XY coordinates for the treasure chests, you'll need an algorithm to find out which treasure chest is closest.

An Algorithm for Finding the Closest Treasure Chest

The x and y parameters are integers (say, 3 and 2), and together they represent the location on the game board where the player guessed. The chests variable will have a value such as [[5, 0], [0, 2], [4, 2]]. That value represents the locations of three treasure chests. You can visualize it as the picture in Figure 13-3. The distances form "rings" around the sonar device located at 3, 2 as in Figure 13-4.
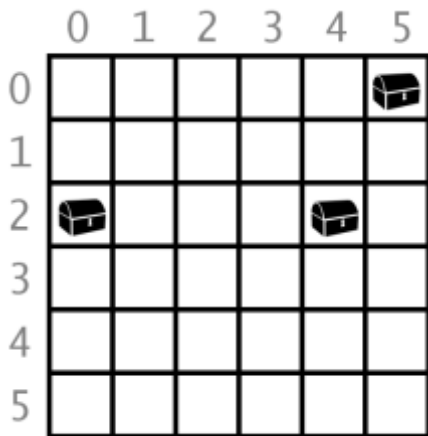

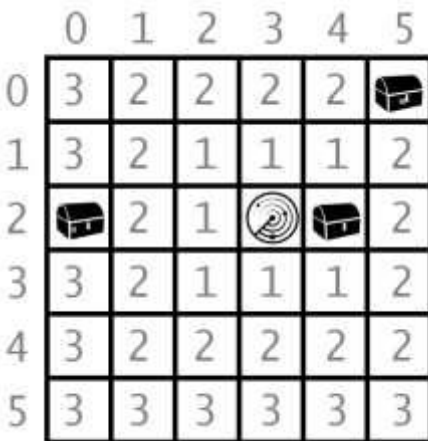Figure 13-3: The treasure chests that [[5, 0], [0, 2], [4, 2]] represents.

Figure 13-4: The board marked with distances from the 3, 2 position.

But how do you translate this into code for the game? You need a way to represent the square ring distance as an expression. Notice that the distance from an XY coordinate is always the

larger of two values: the absolute value of the difference of the two X coordinates and the absolute value of the difference of the two Y coordinates.

That means you should subtract the sonar device's X coordinate and a treasure chest's X coordinate, and then take the absolute value of this number. Do the same for the sonar device's Y coordinate and a treasure chest's Y coordinate. The *larger* of these two values is the distance.

For example, consider the sonar's X and Y coordinates are 3 and 2, like in Figure 13-4. The first treasure chest's X and Y coordinates (that is, first in the list `[[5, 0], [0, 2], [4, 2]]`) are `5` and `0`.

1. For the X coordinates, `3 - 5` evaluates to `-2`, and the absolute value of `-2` is `2`.

2. For the Y coordinates, `2 - 1` evaluates to `1`, and the absolute value of `1` is `1`.

3. Comparing the two absolute values `2` and `1`, the larger value is `2`, so `2` should be the distance between the sonar device and the treasure chest at coordinates 5, 1.

We can look at the board in Figure 13-4 and see that this algorithm works, because the treasure chest at 5, 1 is in the sonar device's 2nd ring. Let's quickly compare the other two chests to see if the distances work out correctly also.

Let's find the distance from the sonar device at 3, 2 and the treasure chest at 0, 2:

1. `abs(3 - 0)` evaluates to `3`.

2. `abs(2 - 2)` evaluates to `0`.

3. `3` is larger than `0`, so the distance from the sonar device at 3, 2 and the treasure chest at 0, 2 is `3`.

Let's find the distance from the sonar device at 3, 2 and the last treasure chest at 4, 2:

1. `abs(3 - 4)` evaluates to `1`.

2. `abs(2 - 2)` evaluates to `0`.

3. `1` is larger than `0`, so the distance is `1`.

Looking at Figure 13-4 you can see all three distances worked out correctly. It seems this algorithm works. The distances from the sonar device to the three sunken treasure chests are 2, 3, and 1. On each guess, you want to know the distance from the sonar device to the closest of the three treasure chest distances. To do this, use a variable called `smallestDistance`. Let's look at the code again:

```
71.        smallestDistance = 100 # any chest will be closer than 100.
```

```
72.      for cx, cy in chests:
73.          if abs(cx - x) > abs(cy - y):
74.              distance = abs(cx - x)
75.          else:
76.              distance = abs(cy - y)
77.
78.          if distance < smallestDistance: # we want the closest treasure
chest.
79.              smallestDistance = distance
```

Line 72 uses the multiple assignment trick in a for loop. For example, the assignment statement spam, eggs = [5, 10] will assign 5 to spam and 10 to eggs.

Because chests is a list where each item in the list is itself a list of two integers, the first of these integers is assigned to cx and the second integer is assigned to cy. So if chests has the value [[5, 0], [0, 2], [4, 2]], cx will have the value 5 and cy will have the value 0 on the first iteration through the loop.

Line 73 determines which is larger: the absolute value of the difference of the X coordinates, or the absolute value of the difference of the Y coordinates. abs(cx - x) > abs(cy - y) seems like much shorter way to say that, doesn't it? Lines 73 to 76 assign the larger of the values to the distance variable.

So on each iteration of the for loop, the distance variable holds the treasure chest's distance from the sonar device. But you want the smallest distance of all the treasure chests. This is where the smallestDistance variable comes in. Whenever the distance variable is smaller than smallestDistance, then the value in distance becomes the new value of smallestDistance.

Give smallestDistance the impossibly high value of 100 at the beginning of the loop so that at least one of the treasure chests you found will be put into smallestDistance. By the time the for loop has finished, you know that smallestDistance holds the shortest distance between the sonar device and all of the treasure chests in the game.

The remove() List Method

The remove() list method will remove the first occurrence of a value matching the passed in argument. For example, try entering the following into the interactive shell:

```
>>> x = [42, 5, 10, 42, 15, 42]
>>> x.remove(10)
>>> x
[42, 5, 42, 15, 42]
```

The 10 value has been removed from the x list. The remove() method removes the first occurrence of the value you pass it, and only the first. For example, type the following into the interactive shell:

```
>>> x = [42, 5, 10, 42, 15, 42]
>>> x.remove(42)
>>> x
[5, 10, 42, 15, 42]
```

Notice that only the first 42 value was removed, but the second and third ones are still there. The remove() method will cause an error if you try to remove a value that isn't in the list:

```
>>> x = [5, 42]
>>> x.remove(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

```
81.        if smallestDistance == 0:
82.            # xy is directly on a treasure chest!
83.            chests.remove([x, y])
84.            return 'You have found a sunken treasure chest!'
```

The only time that smallestDistance is equal to 0 is when the sonar device's XY coordinates are the same as a treasure chest's XY coordinates. This means the player has correctly guessed the location of a treasure chest. Remove this chest's two-integer list from the chests data structure with the remove() list method. Then the function returns 'You have found a sunken treasure chest!'.

```
85.        else:
86.            if smallestDistance < 10:
87.                board[x][y] = str(smallestDistance)
88.                return 'Treasure detected at a distance of %s from the sonar
device.' % (smallestDistance)
89.            else:
90.                board[x][y] = 'O'
91.                return 'Sonar did not detect anything. All treasure chests out
of range.'
```

The else-block starting on line 86 executes if smallestDistance was not 0, which means the player didn't guess an exact location of a treasure chest. If the sonar device's distance was less than 10, line 87 marks the board with the string version of smallestDistance. If not, mark the board with a 'O'.

Getting the Player's Move

```
 94. def enterPlayerMove():
 95.     # Let the player type in their move. Return a two-item list of int xy
coordinates.
 96.     print('Where do you want to drop the next sonar device? (0-59 0-14)
(or type quit)')
 97.     while True:
 98.         move = input()
 99.         if move.lower() == 'quit':
100.             print('Thanks for playing!')
101.             sys.exit()
```

The `enterPlayerMove()` function collects the XY coordinates of the player's next move. The `while` loop will keep asking the player for their next move until they enter a valid move. The player can also type in `'quit'` to quit the game. In that case, line 101 calls the `sys.exit()` function to terminate the program immediately.

```
103.         move = move.split()
104.         if len(move) == 2 and move[0].isdigit() and move[1].isdigit() and
isValidMove(int(move[0]), int(move[1])):
105.             return [int(move[0]), int(move[1])]
106.         print('Enter a number from 0 to 59, a space, then a number from 0
to 14.')
```

Assuming the player has not typed in `'quit'`, the code must ensure it is a valid move: two integers separated by a space. Line 103 calls the `split()` method on `move` as the new value of `move`.

If the player typed in a value like `'1 2 3'`, then the list returned by `split()` would be `['1', '2', '3']`. In that case, the expression `len(move) == 2` would be `False` and the entire expression evaluates immediately to `False`. Python doesn't check the rest of the expression because of short-circuiting (which was described in Chapter 10).

If the list's length is `2` then the two values will be at indexes `move[0]` and `move[1]`. To check if those values are numeric digits (like `'2'` or `'17'`), you could use a function like `isOnlyDigits()` from Chapter 11. But Python already has a function that does this.

The string method `isdigit()` returns `True` if the string consists solely of numbers. Otherwise it returns `False`. Try entering the following into the interactive shell:

```
>>> '42'.isdigit()
True
>>> 'forty'.isdigit()
```

```
False
>>> ''.isdigit()
False
>>> 'hello'.isdigit()
False
>>> x = '10'
>>> x.isdigit()
True
```

Both `move[0].isdigit()` and `move[1].isdigit()` must be `True` for the whole condition to be `True`. The final part of line 104's condition calls the `isValidMove()` function to check if the XY coordinates exist on the board.

If the entire condition is `True`, line 105 returns a two-integer list of the XY coordinates. Otherwise, the execution loops and the player will be asked to enter coordinates again.

Asking the Player to Play Again

```
109. def playAgain():
110.     # This function returns True if the player wants to play again,
otherwise it returns False.
111.     print('Do you want to play again? (yes or no)')
112.     return input().lower().startswith('y')
```

The `playAgain()` function is similar to the `playAgain()` functions in previous chapters.

Printing the Game Instructions for the Player

```
115. def showInstructions():
116.     print('''Instructions:
117. You are the captain of the Simon, a treasure-hunting ship. Your current
mission
118. is to find the three sunken treasure chests that are lurking in the part
of the
119. ocean you are in and collect them.
120.
121. To play, enter the coordinates of the point in the ocean you wish to drop
a
122. sonar device. The sonar can find out how far away the closest chest is to
it.
123. For example, the d below marks where the device was dropped, and the 2's
124. represent distances of 2 away from the device. The 4's represent
125. distances of 4 away from the device.
126.
127.     444444444
```

```
128.     4       4
129.     4 22222 4
130.     4 2   2 4
131.     4 2 d 2 4
132.     4 2   2 4
133.     4 22222 4
134.     4       4
135.     444444444
136. Press enter to continue...''')
137.     input()
```

The showInstructions() is a couple of print() calls that print multi-line strings. The input() function gives the player a chance to press ENTER before printing the next string. This is because the IDLE window can only show so much text at a time.

```
139.     print('''For example, here is a treasure chest (the c) located a
distance of 2 away
140. from the sonar device (the d):
141.
142.     22222
143.     c   2
144.     2 d 2
145.     2   2
146.     22222
147.
148. The point where the device was dropped will be marked with a 2.
149.
150. The treasure chests don't move around. Sonar devices can detect treasure
151. chests up to a distance of 9. If all chests are out of range, the point
152. will be marked with 0
153.
154. If a device is directly dropped on a treasure chest, you have discovered
155. the location of the chest, and it will be collected. The sonar device will
156. remain there.
157.
158. When you collect a chest, all sonar devices will update to locate the next
159. closest sunken treasure chest.
160. Press enter to continue...''')
161.     input()
162.     print()
```

After the player presses ENTER, the function returns.

The Start of the Game

```
165. print('S O N A R !')
166. print()
167. print('Would you like to view the instructions? (yes/no)')
168. if input().lower().startswith('y'):
169.     showInstructions()
```

The expression input().lower().startswith('y') asks the player if they want to see the instructions, and evaluates to True if the player typed in a string that began with 'y' or 'Y'. If so, showInstructions() is called. Otherwise, the game begins.

```
171. while True:
172.     # game setup
173.     sonarDevices = 16
174.     theBoard = getNewBoard()
175.     theChests = getRandomChests(3)
176.     drawBoard(theBoard)
177.     previousMoves = []
```

Line 171's while loop is the main loop for the program. Several variables are set up on lines 173 to 177 and are described in Table 13-1.

Table 13-1: Variables used in the main game loop.

Variable	Description
sonarDevices	The number of sonar devices (and turns) the player has left.
theBoard	The board data structure used for this game.
theChests	The list of chest data structures. getRandomChests() will return a list of three treasure chests at random places on the board.
previousMoves	A list of all the XY moves that the player has made in the game.

Displaying the Game Status for the Player

```
179.     while sonarDevices > 0:
180.         # Start of a turn:
181.
182.         # show sonar device/chest status
```

```
183.         if sonarDevices > 1: extraSsonar = 's'
184.         else: extraSsonar = ''
185.         if len(theChests) > 1: extraSchest = 's'
186.         else: extraSchest = ''
187.         print('You have %s sonar device%s left. %s treasure chest%s
remaining.' % (sonarDevices, extraSsonar, len(theChests), extraSchest))
```

Line 179's `while` loop executes as long as the player has sonar devices remaining. Line 187 prints a message telling the user how many sonar devices and treasure chests are left. But there's a small problem.

If there are two or more sonar devices left, you want to print `'2 sonar devices'`. But if there's only one sonar device left, you want to print `'1 sonar device'` left. You only want the plural form of "devices" if there are multiple sonar devices. The same goes for `'2 treasure chests'` and `'1 treasure chest'`.

Lines 183 through 186 have code after the `if` and `else` statements' colon. This is perfectly valid Python. Instead of having a block of code after the statement, you can use the rest of the same line to make your code more concise.

The two variables named `extraSsonar` and `extraSchest` are set to `'s'` (space) if there are multiple sonar devices or treasures chests. Otherwise, they are blank strings. These variables are used on line 187.

Getting the Player's Move

```
189.         x, y = enterPlayerMove()
190.         previousMoves.append([x, y]) # we must track all moves so that
sonar devices can be updated.
191.         moveResult = makeMove(theBoard, theChests, x, y)
192.         if moveResult == False:
193.             continue
```

Line 189 uses multiple assignment since `enterPlayerMove()` returns a two-item list. The first item in the returned list is assigned to the `x` variable. The second is assigned to the `y` variable.

They are then appended to the end of the `previousMoves` list. This means `previousMoves` is a list of XY coordinates of each move the player makes in this game. This list is used later in the program on line 198.

The `x`, `y`, `theBoard`, and `theChests` variables are all passed to the `makeMove()` function. This function will make the necessary modifications to the game board to place a sonar device on the board.

If makeMove() returns the value False, then there was a problem with the x and y values you passed it. The continue statement will send the execution back to the start of the while loop on line 179 to ask the player for XY coordinates again.

Finding a Sunken Treasure Chest

```
195.          else:
196.              if moveResult == 'You have found a sunken treasure chest!':
197.                  # update all the sonar devices currently on the map.
198.                  for x, y in previousMoves:
199.                      makeMove(theBoard, theChests, x, y)
200.              drawBoard(theBoard)
201.              print(moveResult)
```

If makeMove() didn't return the value False, it would have returned a string of the results of that move. If this string was 'You have found a sunken treasure chest!', then all the sonar devices on the board should be updated to detect the *next* closest treasure chest on the board. The XY coordinates of all the sonar devices are in previousMoves. By iterating over previousMoves on line 198, you can pass all of these XY coordinates to the makeMove() function again to redraw the values on the board.

Because the program doesn't print anything new here, the player doesn't realize the program is redoing all of the previous moves. It just appears that the board updates itself.

Checking if the Player has Won

```
203.          if len(theChests) == 0:
204.              print('You have found all the sunken treasure chests!
Congratulations and good game!')
205.              break
```

Remember that the makeMove() function modifies the theChests list you sent it. Because theChests is a list, any changes made to it inside the function will persist after execution returns from the function. makeMove() removes items from theChests when treasure chests are found, so eventually (if the player keeps guessing correctly) all of the treasure chests will have been removed. Remember, by "treasure chest" we mean the two-item lists of the XY coordinates inside the theChests list.

When all the treasure chests have been found on the board and removed from theChests, the theChests list will have a length of 0. When that happens, display a congratulations to the player, and then execute a break statement to break out of this while loop. Execution will then move to line 209, the first line after the while-block.

Checking if the Player has Lost

```
207.          sonarDevices -= 1
```

Line 207 is the last line of the while loop that started on line 179. Decrement the sonarDevices variable because the player has used one. If the player keeps missing the treasure chests, eventually sonarDevices will be reduced to 0. After this line, execution jumps back up to line 179 so it can re-evaluate the while statement's condition (which is sonarDevices > 0).

If sonarDevices is 0, then the condition will be False and execution will continue outside the while-block on line 209. But until then, the condition will remain True and the player can keep making guesses.

```
209.     if sonarDevices == 0:
210.         print('We\'ve run out of sonar devices! Now we have to turn the
ship around and head')
211.         print('for home with treasure chests still out there! Game over.')
212.         print('    The remaining chests were here:')
213.         for x, y in theChests:
214.             print('    %s, %s' % (x, y))
```

Line 209 is the first line outside the while loop. When the execution reaches this point the game is over. If sonarDevices is 0, you know the player ran out of sonar devices before finding all the chests and lost.

Lines 210 to 212 will tell the player they've lost. The for loop on line 213 will go through the treasure chests remaining in theChests and show their location to the player so that they can know where the treasure chests had been lurking.

The sys.exit() Function

```
216.     if not playAgain():
217.         sys.exit()
```

Win or lose, playAgain() is called again to let the player type in whether they want to keep playing or not. If not, then playAgain() returns False. The not operator on line 216 changes this to True, making the if statement's condition True and the sys.exit() function is executed. This will cause the program to terminate.

Otherwise, execution jumps back to the beginning of the while loop on line 171 and a new game begins.

Summary

Remember how our Tic Tac Toe game numbered the spaces on the Tic Tac Toe board 1 through 9? This sort of coordinate system might have been okay for a board with less than ten spaces. But the Sonar board has 900 spaces! The Cartesian coordinate system we learned in the last chapter really makes all these spaces manageable, especially when our game needs to find the distance between two points on the board.

Locations in games that use a Cartesian coordinate system can be stored in a list of lists so that the first index is the X-coordinate and the second index is the Y-coordinate. This make accessing a coordinates look like `board[x][y]`.

These data structures (such as the ones used for the ocean and locations of the treasure chests) make it possible to have complicated concepts represented as data, and your game programs become mostly about modifying these data structures.

In the next chapter, we will be representing letters as numbers using their ASCII numbers. (This is the same ASCII term we used in "ASCII art" previously.) By representing text as numbers, we can perform math operations on them which will encrypt or decrypt secret messages.

Chapter 14

CAESAR CIPHER

> Topics Covered In This Chapter:
> - Cryptography and ciphers
> - Encrypting and decrypting
> - Ciphertext, plaintext, keys, and symbols
> - The Caesar Cipher
> - ASCII ordinal values
> - The chr() and ord() functions
> - The isalpha() string method
> - The isupper() and islower() string methods
> - Cryptanalysis
> - The brute force technique

The program in this chapter isn't really a game, but it is a fun program. The program will convert normal English into a secret code. It can also convert secret codes back into regular English again. Only someone who is knowledgeable about secret codes will be able to understand our secret messages.

Because this program manipulates text to convert it into secret messages, you will learn several new functions and methods for manipulating strings. You will also learn how programs can do math with text strings just as it can with numbers.

Cryptography

The science of writing secret codes is called **cryptography**. For thousands of years cryptography has made secret messages that only the sender and recipient could read, even if someone captured the messenger and read the coded message. A secret code system is called a **cipher**. The cipher used by the program in this chapter is called the Caesar cipher.

In cryptography, we call the message that we want to be secret the **plaintext**. The plaintext could look like this:

```
Hello there! The keys to the house are hidden under the flower pot.
```

Converting the plaintext into the encoded message is called **encrypting** the plaintext. The plaintext is encrypted into the **ciphertext**. The ciphertext looks like random letters, and we cannot

understand what the original plaintext was just by looking at the ciphertext. Here is the previous example encrypted into ciphertext:

```
Yvccf kyviv! Kyv bvpj kf kyv yfljv riv yzuuve leuvi kyv wcfnvi gfk.
```

But if you know about the cipher used to encrypt the message, you can **decrypt** the ciphertext back to the plaintext. (Decryption is the opposite of encryption.)

Many ciphers also use keys. **Keys** are secret values that let you decrypt ciphertext that was encrypted using a specific cipher. Think of the cipher as being like a door lock. You can only unlock it with a particular key.

If you are interested in writing cryptography programs, you can read my other book, "Hacking Secret Ciphers with Python". It is free to download from http://inventwithpython.com/hacking.

The Caesar Cipher

The key for the Caesar Cipher will be a number from 1 to 26. Unless you know the key (that is, know the number used to encrypt the message), you won't be able to decrypt the secret code.

The **Caesar Cipher** was one of the earliest ciphers ever invented. In this cipher, you encrypt a message by taking each letter in the message (in cryptography, these letters are called **symbols** because they can be letters, numbers, or any other sign) and replacing it with a "shifted" letter. If you shift the letter A by one space, you get the letter B. If you shift the letter A by two spaces, you get the letter C. Figure 14-1 is a picture of some letters shifted over by three spaces.
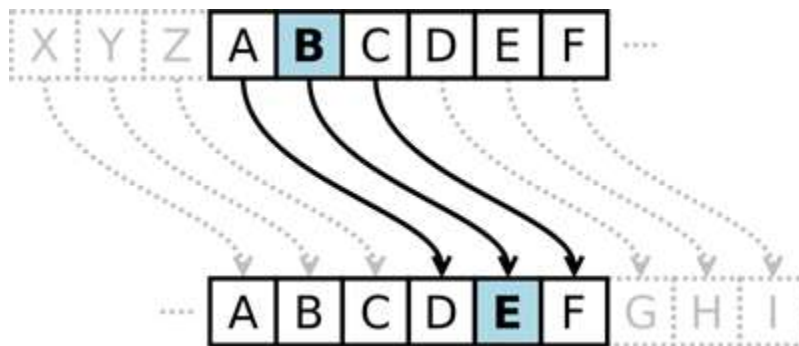
Figure 14-1: Shifting over letters by three spaces. Here, B becomes E.

To get each shifted letter, draw out a row of boxes with each letter of the alphabet. Then draw a second row of boxes under it, but start a certain number (this number is the key) of spaces over. After the letters at the end, wrap around back to the start of the boxes. Here is an example with the letters shifted by three spaces:

Alphabet shifted by 3 spaces.
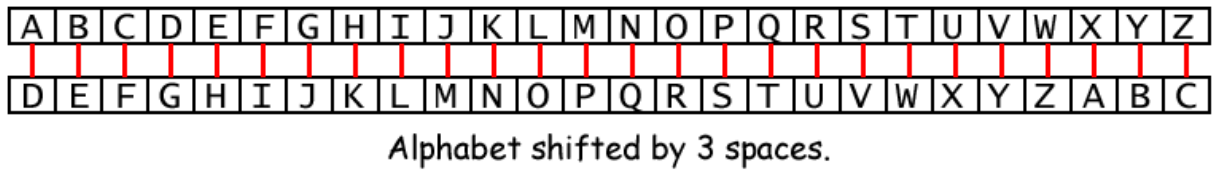
Figure 14-2: The entire alphabet shifted by three spaces.

The number of spaces you shift is the key in the Caesar Cipher. The example above shows the letter translations for the key 3.

If you encrypt the plaintext "Howdy" with a key of 3, then:

- The "H" becomes "K".

- The letter "o" becomes "r".

- The letter "w" becomes "z".

- The letter "d" becomes "g".

- The letter "y" becomes "b".

The ciphertext of "Hello" with key 3 becomes "Krzgb".

We will keep any non-letter characters the same. To decrypt "Krzgb" with the key 3, we go from the bottom boxes back to the top:

- The letter "K" becomes "H".

- The letter "r" becomes "o".

- The letter "z" becomes "w".

- The letter "g" becomes "d".

- The letter "b" becomes "y".

ASCII, and Using Numbers for Letters

How do we implement this shifting of the letters as code? We can do this by representing each letter as a number called an **ordinal**, and then adding or subtracting from this number to form a new ordinal (and a new letter). ASCII (pronounced "ask-ee" and stands for American Standard Code for Information Interchange) is a code that connects each character to a number between 32 and 126.

The capital letters "A" through "Z" have the ASCII numbers 65 through 90. The lowercase letters "a" through "z" have the ASCII numbers 97 through 122. The numeric digits "0" through "9" have the ASCII numbers 48 through 57. Table 14-1 shows all the ASCII characters and ordinals.

Modern computers use UTF-8 instead of ASCII. But UTF-8 is backwards compatible with ASCII, so the UTF-8 ordinals for ASCII characters are the same as ASCII's ordinals.

Table 14-1: The ASCII Table

32	(space)	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124		
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		

So if you wanted to shift "A" by three spaces, you would do the following:

- Convert "A" to an ordinal (65).

- Add 3 to 65, to get 68.

- Convert the ordinal 68 back to a letter ("D").

The chr() and ord() functions can convert between characters and ordinals.

The chr() and ord() Functions

The chr() function (pronounced "char", short for "character") takes an integer ordinal and returns a single-character string. The ord() function (short for "ordinal") takes a single-character string, and returns the integer ordinal value. Try entering the following into the interactive shell:

```
>>> chr(65)
'A'
```

```
>>> ord('A')
65
>>> chr(65+8)
'I'
>>> chr(52)
'4'
>>> chr(ord('F'))
'F'
>>> ord(chr(68))
68
```

On the third line, chr(65+8) evaluates to chr(73). If you look at the ASCII table, you can see that 73 is the ordinal for the capital letter "I".

On the fifth line, chr(ord('F')) evaluates to chr(70) which evaluates to 'F'. The ord() and chr() functions are the opposite of each other.

Sample Run of Caesar Cipher

Here is a sample run of the Caesar Cipher program, encrypting a message:

```
Do you wish to encrypt or decrypt a message?
encrypt
Enter your message:
The sky above the port was the color of television, tuned to a dead channel.
Enter the key number (1-26)
13
Your translated text is:
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb n qrnq punaary.
```

Now run the program and decrypt the text that you just encrypted.

```
Do you wish to encrypt or decrypt a message?
decrypt
Enter your message:
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb n qrnq punaary.
Enter the key number (1-26)
13
Your translated text is:
The sky above the port was the color of television, tuned to a dead channel.
```

If you do not decrypt with the correct key, the decrypted text will be garbage data:

```
Do you wish to encrypt or decrypt a message?
decrypt
Enter your message:
```

```
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb n qrnq punaary.
Enter the key number (1-26)
15
Your translated text is:
Rfc qiw yzmtc rfc nmpr uyq rfc amjmp md rcjctgqgml, rslcb rm y bcyb afyllcj.
```

Source Code of Caesar Cipher

Here is the source code for the Caesar Cipher program. After you type this code in, save the file as *cipher.py*. If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/cipher.

caesar.py

```python
1. # Caesar Cipher
2.
3. MAX_KEY_SIZE = 26
4.
5. def getMode():
6.     while True:
7.         print('Do you wish to encrypt or decrypt a message?')
8.         mode = input().lower()
9.         if mode in 'encrypt e decrypt d'.split():
10.             return mode
11.         else:
12.             print('Enter either "encrypt" or "e" or "decrypt" or "d".')
13.
14. def getMessage():
15.     print('Enter your message:')
16.     return input()
17.
18. def getKey():
19.     key = 0
20.     while True:
21.         print('Enter the key number (1-%s)' % (MAX_KEY_SIZE))
22.         key = int(input())
23.         if (key >= 1 and key <= MAX_KEY_SIZE):
24.             return key
25.
26. def getTranslatedMessage(mode, message, key):
27.     if mode[0] == 'd':
28.         key = -key
29.     translated = ''
30.
31.     for symbol in message:
32.         if symbol.isalpha():
```

```
33.                num = ord(symbol)
34.                num += key
35.
36.                if symbol.isupper():
37.                    if num > ord('Z'):
38.                        num -= 26
39.                    elif num < ord('A'):
40.                        num += 26
41.                elif symbol.islower():
42.                    if num > ord('z'):
43.                        num -= 26
44.                    elif num < ord('a'):
45.                        num += 26
46.
47.                translated += chr(num)
48.          else:
49.                translated += symbol
50.      return translated
51.
52. mode = getMode()
53. message = getMessage()
54. key = getKey()
55.
56. print('Your translated text is:')
57. print(getTranslatedMessage(mode, message, key))
```

How the Code Works

The encryption and decryption processes are the reverse of the other, and even then they still share much of the same code. Let's look at how each line works.

```
1. # Caesar Cipher
2.
3. MAX_KEY_SIZE = 26
```

The first line is just a comment. MAX_KEY_SIZE is a constant that stores the integer 26 in it. MAX_KEY_SIZE reminds us that in this program, the key used in the cipher should be between 1 and 26.

Deciding to Encrypt or Decrypt

```
5. def getMode():
6.     while True:
7.         print('Do you wish to encrypt or decrypt a message?')
```

```
8.          mode = input().lower()
9.          if mode in 'encrypt e decrypt d'.split():
10.             return mode
11.         else:
12.             print('Enter either "encrypt" or "e" or "decrypt" or "d".')
```

The `getMode()` function will let the user type in if they want encryption or decryption mode for the program. The value returned from `input()` and `lower()` is stored in `mode`. The `if` statement's condition checks if the string stored in `mode` exists in the list returned by `'encrypt e decrypt d'.split()`.

This list is `['encrypt', 'e', 'decrypt', 'd']`, but it is easier for the programmer to type `'encrypt e decrypt d'.split()` and not type in all those quotes and commas. Use whichever is easiest for you; they both evaluate to the same list value.

This function will return the string in `mode` as long as `mode` is equal to `'encrypt'`, `'e'`, `'decrypt'`, or `'d'`. Therefore, `getMode()` will return the string `'e'` or the string `'d'` (but the user can type in either "e", "encrypt", "d", or "decrypt".)

Getting the Message from the Player

```
14. def getMessage():
15.     print('Enter your message:')
16.     return input()
```

The `getMessage()` function simply gets the message to encrypt or decrypt from the user and returns it.

Getting the Key from the Player

```
18. def getKey():
19.     key = 0
20.     while True:
21.         print('Enter the key number (1-%s)' % (MAX_KEY_SIZE))
22.         key = int(input())
23.         if (key >= 1 and key <= MAX_KEY_SIZE):
24.             return key
```

The `getKey()` function lets the player type in the key they will use to encrypt or decrypt the message. The `while` loop ensures that the function keeps looping until the user enters a valid key.

A valid key here is one that is between the integer values 1 and 26 (remember that MAX_KEY_SIZE will only ever have the value 26 because it is constant). It then returns this key. Line 22 sets key to the integer version of what the user typed in, so getKey() returns an integer.

Encrypt or Decrypt the Message with the Given Key

```
26. def getTranslatedMessage(mode, message, key):
27.     if mode[0] == 'd':
28.         key = -key
29.     translated = ''
```

getTranslatedMessage() does the encrypting and decrypting. It has three parameters:

- mode sets the function to encryption mode or decryption mode.

- message is the plaintext (or ciphertext) to be encrypted (or decrypted).

- key is the key that is used in this cipher.

Line 27 checks if the first letter in the mode variable is the string 'd'. If so, then the program is in decryption mode. The only difference between the decryption and encryption mode is that in decryption mode the key is set to the negative version of itself. If key was the integer 22, then in decryption mode set it to -22. The reason why will be explained later.

translated is the string of the result: either the ciphertext (if you are encrypting) or the plaintext (if you are decrypting). It starts as the blank string and has encrypted or decrypted characters concatenated to the end of it.

The isalpha() String Method

The isalpha() string method will return True if the string is an uppercase or lowercase letter from A to Z. If the string contains any non-letter characters, then isalpha() will return False. Try entering the following into the interactive shell:

```
>>> 'Hello'.isalpha()
True
>>> 'Forty two'.isalpha()
False
>>> 'Fortytwo'.isalpha()
True
>>> '42'.isalpha()
False
>>> ''.isalpha()
False
```

As you can see, `'Forty two'.isalpha()` will return `False` because `'Forty two'` has a space in it, which is a non-letter character. `'Fortytwo'.isalpha()` returns `True` because it doesn't have this space.

`'42'.isalpha()` returns `False` because both `'4'` and `'2'` are non-letter characters. `isalpha()` only returns `True` if the string has only letter characters and isn't blank.

The `isalpha()` method is used in the next few lines of the program.

```
31.      for symbol in message:
32.          if symbol.isalpha():
33.              num = ord(symbol)
34.              num += key
```

Line 31's `for` loop iterates over each letter (in cryptography they are called symbols) in the `message` string. On each iteration through this loop, `symbol` will have the value of a letter in `message`.

Line 32 is there because only letters will be encrypted or decrypted. Numbers, punctuation marks, and everything else will stay in their original form. The `num` variable will hold the integer ordinal value of the letter stored in `symbol`. Line 34 then "shifts" the value in `num` by the value in `key`.

The `isupper()` and `islower()` String Methods

The `isupper()` and `islower()` string methods (which are on line 36 and 41) work in a way that is similar to the `isdigit()` and `isalpha()` methods.

`isupper()` will return `True` if the string it is called on contains at least one uppercase letter and no lowercase letters. `islower()` returns `True` if the string it is called on contains at least one lowercase letter and no uppercase letters. Otherwise these methods return `False`.

Try entering the following into the interactive shell:

```
>>> 'HELLO'.isupper()
True
>>> 'hello'.isupper()
False
>>> 'hello'.islower()
True
>>> 'Hello'.islower()
False
>>> 'LOOK OUT BEHIND YOU!'.isupper()
True
>>> '42'.isupper()
False
```

```
>>> '42'.islower()
False
>>> ''.isupper()
False
>>> ''.islower()
False
```

Encrypting or Decrypting Each Letter

```
36.             if symbol.isupper():
37.                 if num > ord('Z'):
38.                     num -= 26
39.                 elif num < ord('A'):
40.                     num += 26
```

Line 36 checks if the symbol is an uppercase letter. If so, there are two special cases to worry about. What if symbol was 'Z' and key was 4? If that were the case, the value of num here would be the character '^' (The ordinal of '^' is 94). But ^ isn't a letter at all. You want the ciphertext to "wrap around" to the beginning of the alphabet.

Check if num has a value larger than the ordinal value for "Z". If so, then **subtract** 26 (because there are 26 letters in total) from num. After doing this, the value of num is 68. 68 is the correct ordinal value for 'D'.

```
41.             elif symbol.islower():
42.                 if num > ord('z'):
43.                     num -= 26
44.                 elif num < ord('a'):
45.                     num += 26
```

If the symbol is a lowercase letter, the program runs code that is similar to lines 36 through 40. The only difference is that it uses ord('z') and ord('a') instead of ord('Z') and ord('A').

In decrypting mode, then key would be negative. The special case would be where num -= 26 is less than the ASCII value for "a". In that case, **add** 26 to num to have it "wrap around" to the end of the alphabet.

```
47.             translated += chr(num)
48.         else:
49.             translated += symbol
```

Line 47 concatenates the encrypted/decrypted character to the translated string.

If the symbol was not an uppercase or lowercase letter, then line 48 concatenates the original symbol to the translated string. Therefore, spaces, numbers, punctuation marks, and other characters won't be encrypted or decrypted.

```
50.      return translated
```

The last line in the getTranslatedMessage() function returns the translated string.

The Start of the Program

```
52. mode = getMode()
53. message = getMessage()
54. key = getKey()
55. print('Your translated text is:')
56. print(getTranslatedMessage(mode, message, key))
```

The start of the program calls each of the three functions defined previously to get the mode, message, and key from the user. These three values are passed to getTranslatedMessage() whose return value (the translated string) is printed to the user.

Brute Force

That's the entire Caesar Cipher. However, while this cipher may fool some people who don't understand cryptography, it won't keep a message secret from someone who knows cryptanalysis. While cryptography is the science of making codes, **cryptanalysis** is the science of breaking codes.

```
Do you wish to encrypt or decrypt a message?
encrypt
Enter your message:
Doubts may not be pleasant, but certainty is absurd.
Enter the key number (1-26)
8
Your translated text is:
Lwcjba uig vwb jm xtmiaivb, jcb kmzbiqvbg qa ijaczl.
```

The whole point of cryptography is that so if someone else gets their hands on the encrypted message, they cannot figure out the original unencrypted message from it. Let's pretend we are the code breaker and all we have is the encrypted text:

```
Lwcjba uig vwb jm xtmiaivb, jcb kmzbiqvbg qa ijaczl.
```

Brute force is the technique of trying every possible key until you find the correct one. Because there are only 26 possible keys, it would be easy for a cryptanalyst to write a hacking program than decrypts with every possible key. Then they could look for the key that decrypts to plain English. Let's add a brute force feature to the program.

Adding the Brute Force Mode

First, change lines 7, 9, and 12 (which are in the getMode() function) to look like the following (the changes are in bold):

```
 5. def getMode():
 6.     while True:
 7.         print('Do you wish to encrypt or decrypt or brute force a
message?')
 8.         mode = input().lower()
 9.         if mode in 'encrypt e decrypt d brute b'.split():
10.             return mode[0]
11.         else:
12.             print('Enter either "encrypt" or "e" or "decrypt" or "d" or
"brute" or "b".')
```

This code will let the user select "brute force" as a mode. Modify and add the following changes to the main part of the program:

```
52. mode = getMode()
53. message = getMessage()
54. if mode[0] != 'b':
55.     key = getKey()
56.
57. print('Your translated text is:')
58. if mode[0] != 'b':
59.     print(getTranslatedMessage(mode, message, key))
60. else:
61.     for key in range(1, MAX_KEY_SIZE + 1):
62.         print(key, getTranslatedMessage('decrypt', message, key))
```

These changes ask the user for a key if they are not in "brute force" mode. The original getTranslatedMessage() call is made and the translated string is printed.

However, if the user is in "brute force" mode then getTranslatedMessage() loop that iterates from 1 all the way up to MAX_KEY_SIZE (which is 26). Remember that when the range() function returns a list of integers up to, but not including, the second parameter, which is why you have +

1. This program will print every possible translation of the message (including the key number used in the translation). Here is a sample run of this modified program:

```
Do you wish to encrypt or decrypt or brute force a message?
brute
Enter your message:
Lwcjba uig vwb jm xtmiaivb, jcb kmzbiqvbg qa ijaczl.
Your translated text is:
1 Kvbiaz thf uva il wslhzhua, iba jlyahpuaf pz hizbyk.
2 Juahzy sge tuz hk vrkgygtz, haz ikxzgotze oy ghyaxj.
3 Itzgyx rfd sty gj uqjfxfsy, gzy hjwyfnsyd nx fgxzwi.
4 Hsyfxw qec rsx fi tpiewerx, fyx givxemrxc mw efwyvh.
5 Grxewv pdb qrw eh sohdvdqw, exw fhuwdlqwb lv devxug.
6 Fqwdvu oca pqv dg rngcucpv, dwv egtvckpva ku cduwtf.
7 Epvcut nbz opu cf qmfbtbou, cvu dfsubjouz jt bctvse.
8 Doubts may not be pleasant, but certainty is absurd.
9 Cntasr lzx mns ad okdzrzms, ats bdqszhmsx hr zartqc.
10 Bmszrq kyw lmr zc njcyqylr, zsr acpryglrw gq yzqspb.
11 Alryqp jxv klq yb mibxpxkq, yrq zboqxfkqv fp xyproa.
12 Zkqxpo iwu jkp xa lhawowjp, xqp yanpwejpu eo wxoqnz.
13 Yjpwon hvt ijo wz kgzvnvio, wpo xzmovdiot dn vwnpmy.
14 Xiovnm gus hin vy jfyumuhn, von wylnuchns cm uvmolx.
15 Whnuml ftr ghm ux iextltgm, unm vxkmtbgmr bl tulnkw.
16 Vgmtlk esq fgl tw hdwsksfl, tml uwjlsaflq ak stkmjv.
17 Uflskj drp efk sv gcvrjrek, slk tvikrzekp zj rsjliu.
18 Tekrji cqo dej ru fbuqiqdj, rkj suhjqydjo yi qrikht.
19 Sdjqih bpn cdi qt eatphpci, qji rtgipxcin xh pqhjgs.
20 Rciphg aom bch ps dzsogobh, pih qsfhowbhm wg opgifr.
21 Qbhogf znl abg or cyrnfnag, ohg pregnvagl vf nofheq.
22 Pagnfe ymk zaf nq bxqmemzf, ngf oqdfmuzfk ue mnegdp.
23 Ozfmed xlj yze mp awpldlye, mfe npceltyej td lmdfco.
24 Nyeldc wki xyd lo zvokckxd, led mobdksxdi sc klcebn.
25 Mxdkcb vjh wxc kn yunjbjwc, kdc lnacjrwch rb jkbdam.
26 Lwcjba uig vwb jm xtmiaivb, jcb kmzbiqvbg qa ijaczl.
```

After looking over each row, you can see that the 8th message isn't garbage, but plain English! The cryptanalyst can deduce that the original key for this encrypted text must have been 8. This brute force would have been difficult to do back in the days of Caesars and the Roman Empire, but today we have computers that can quickly go through millions or even billions of keys in a short time.

Summary

Computers are good at doing mathematics. When we create a system to translate some piece of information into numbers (such as we do with text and ordinals or with space and coordinate systems), computer programs can process these numbers quickly and efficiently.

But while our Caesar cipher program here can encrypt messages that will keep them secret from people who have to figure it out with pencil and paper, it won't keep it secret from people who know how to get computers to process information for them. (Our brute force mode proves this.)

A large part of figuring out how to write a program is figuring out how to represent the information you want to manipulate as values that Python can understand.

The next chapter will present Reversi (also known as Othello). The AI that plays this game will be much more advanced than the AI that played Tic Tac Toe in chapter 9. In fact, the AI is so good most of the time you'll be unable to beat it!

Chapter 15

REVERSI

Topics Covered In This Chapter:
- The `bool()` Function
- How to Play Reversi

In this chapter, we'll make a game called Reversi (also called Othello). Reversi is a board game that is played on a grid, so we'll use a Cartesian coordinate system with XY coordinates. It is a game played with two players. Our version of the game will have a computer AI that is more advanced than the AI we made for Tic Tac Toe. In fact, this AI is so good that it will probably beat you almost every time you play. (I know I lose whenever I play against it!)

Reversi has an 8×8 board and tiles that are black on one side and white on the other (our game will use O's and X's instead). The starting board looks like Figure 15-1. The black player and white player take turns placing down a new tile of their color. Any of the opponent's tiles that are between the new tile and the other tiles of that color are flipped. The goal of the game is to have as many of the tiles with your color as possible. For example, Figure 15-2 is what it looks like if the white player places a new white tile on space 5, 6.
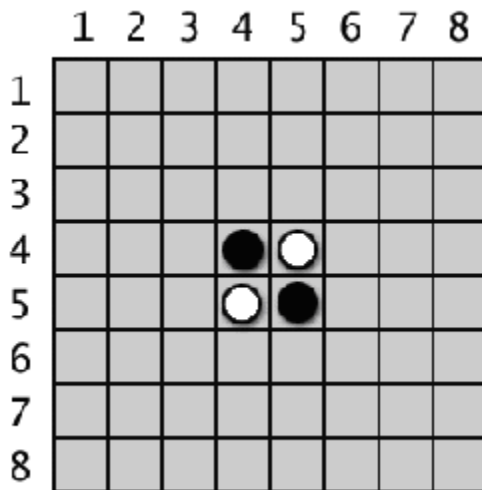
Figure 15-1: The starting Reversi board has two white tiles and two black tiles.
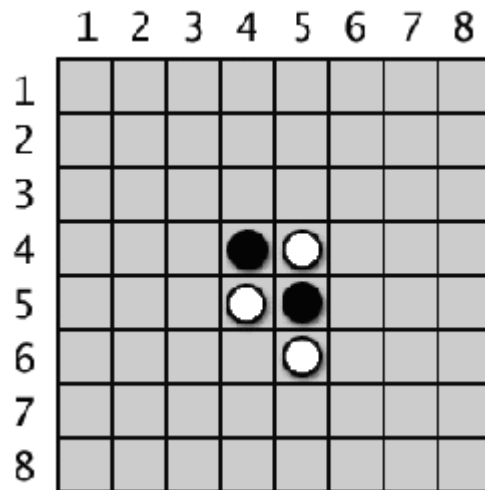
Figure 15-2: White places a new tile.

The black tile at 5, 5 is between the new white tile and the existing white tile at 5, 4. That black tile is flipped over and becomes a new white tile, making the board look like Figure 15-3. Black makes a similar move next, placing a black tile on 4, 6 which flips the white tile at 4, 5. This results in a board that looks like Figure 15-4.
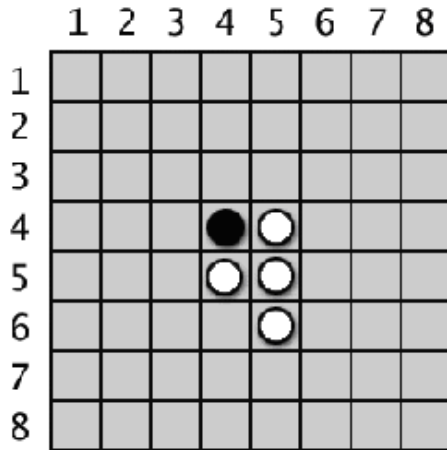
Figure 15-3: White's move will flip over one of black's tiles.
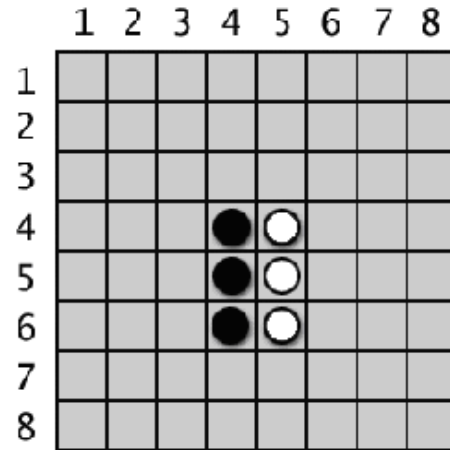
Figure 15-4: Black places a new tile, which flips over one of white's tiles.

Tiles in all directions are flipped as long as they are between the player's new tile and existing tile. In Figure 15-5, the white player places a tile at 3, 6 and flips black tiles in both directions (marked by the lines.) The result is in Figure 15-6.
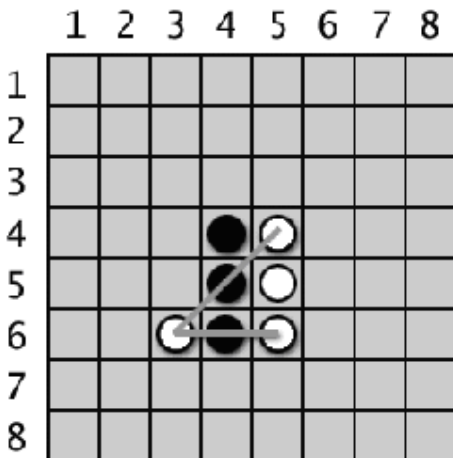
Figure 15-5: White's second move at 3, 6 will flip two of black's tiles.
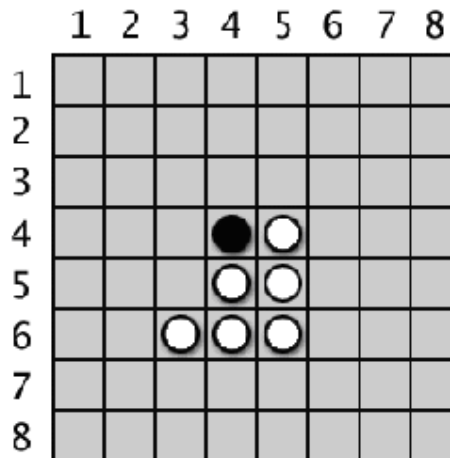
Figure 15-6: The board after white's second move.

Each player can quickly flip many tiles on the board in one or two moves. Players must always make a move that captures at least one tile. The game ends when a player either cannot make a move, or the board is completely full. The player with the most tiles of their color wins.

The AI we make for this game will simply look for any corner moves they can take. If there are no corner moves available, then the computer will select the move that claims the most tiles.

Sample Run of Reversi

```
Welcome to Reversi!
Do you want to be X or O?
x
The player will go first.
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
1 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
2 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
3 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
4 |   |   |   | X | O |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
5 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
6 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
7 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
8 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
```

```
   +---+---+---+---+---+---+---+---+
You have 2 points. The computer has 2 points.
Enter your move, or type quit to end the game, or hints to turn off/on hints.
53
     1   2   3   4   5   6   7   8
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
1  |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
2  |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
3  |   |   |   |   | X |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
4  |   |   | X | X |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
5  |   |   | O | X |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
6  |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
7  |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
8  |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
You have 4 points. The computer has 1 points.
Press Enter to see the computer's move.

...skipped for brevity...

     1   2   3   4   5   6   7   8
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
1  | O | O | O | O | O | O | O | O |
```

```
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
2 | O | O | O | O | O | O | O | O |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
3 | O | O | O | O | O | O | O | O |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
4 | O | O | X | O | O | O | O | O |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
5 | O | O | O | X | O | X | O | X |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
6 | O | X | O | X | X | O | O |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
7 | O | X | X | O | O | O | O | O |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
8 | O | X | X | O |   |   | X |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
You have 12 points. The computer has 48 points.
Enter your move, or type quit to end the game, or hints to turn off/on hints.
86
X scored 15 points. O scored 46 points.
You lost. The computer beat you by 31 points.
Do you want to play again? (yes or no)
no
```

As you can see, the AI was pretty good at beating me 46 to 15. To help the player out, we'll program the game to provide hints. If the player types 'hints' as their move, they can toggle the hints mode on and off. When hints mode is on, all the possible moves the player can make will show up on the board as '.' characters, like this:

```
      1   2   3   4   5   6   7   8
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  1 |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  2 |   |   |   | . |   | . |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  3 |   |   |   | 0 | 0 | 0 |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  4 |   |   | . | 0 | 0 | X |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  5 |   |   | . | 0 | 0 | 0 | X |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  6 |   |   |   | . |   | . |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  7 |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  8 |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
```

Source Code of Reversi

Reversi is a mammoth program compared to our previous games. It's over 300 lines long! But don't worry, many of these lines are comments or blank lines to space out the code and make it more readable.

As with our other programs, we'll first create several functions to carry out Reversi-related tasks that the main section will call. Roughly the first 250 lines of code are for these helper functions, and the last 50 lines of code implement the Reversi game itself.

If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/reversi.

```
                                                              reversi.py
 1. # Reversi
 2.
 3. import random
 4. import sys
 5.
 6. def drawBoard(board):
 7.     # This function prints out the board that it was passed. Returns None.
 8.     HLINE = '   +---+---+---+---+---+---+---+---+'
 9.     VLINE = '   |   |   |   |   |   |   |   |   |'
10.
11.     print('    1   2   3   4   5   6   7   8')
12.     print(HLINE)
13.     for y in range(8):
14.         print(VLINE)
15.         print(y+1, end=' ')
16.         for x in range(8):
17.             print('| %s' % (board[x][y]), end=' ')
18.         print('|')
19.         print(VLINE)
20.         print(HLINE)
21.
22.
23. def resetBoard(board):
24.     # Blanks out the board it is passed, except for the original starting
position.
25.     for x in range(8):
26.         for y in range(8):
27.             board[x][y] = ' '
28.
29.     # Starting pieces:
30.     board[3][3] = 'X'
31.     board[3][4] = 'O'
32.     board[4][3] = 'O'
33.     board[4][4] = 'X'
34.
35.
36. def getNewBoard():
37.     # Creates a brand new, blank board data structure.
38.     board = []
39.     for i in range(8):
40.         board.append([' '] * 8)
41.
```

```
42.     return board
43.
44.
45. def isValidMove(board, tile, xstart, ystart):
46.     # Returns False if the player's move on space xstart, ystart is
invalid.
47.     # If it is a valid move, returns a list of spaces that would become
the player's if they made a move here.
48.     if board[xstart][ystart] != ' ' or not isOnBoard(xstart, ystart):
49.         return False
50.
51.     board[xstart][ystart] = tile # temporarily set the tile on the board.
52.
53.     if tile == 'X':
54.         otherTile = 'O'
55.     else:
56.         otherTile = 'X'
57.
58.     tilesToFlip = []
59.     for xdirection, ydirection in [[0, 1], [1, 1], [1, 0], [1, -1], [0, -
1], [-1, -1], [-1, 0], [-1, 1]]:
60.         x, y = xstart, ystart
61.         x += xdirection # first step in the direction
62.         y += ydirection # first step in the direction
63.         if isOnBoard(x, y) and board[x][y] == otherTile:
64.             # There is a piece belonging to the other player next to our
piece.
65.             x += xdirection
66.             y += ydirection
67.             if not isOnBoard(x, y):
68.                 continue
69.             while board[x][y] == otherTile:
70.                 x += xdirection
71.                 y += ydirection
72.                 if not isOnBoard(x, y): # break out of while loop, then
continue in for loop
73.                     break
74.             if not isOnBoard(x, y):
75.                 continue
76.             if board[x][y] == tile:
77.                 # There are pieces to flip over. Go in the reverse
direction until we reach the original space, noting all the tiles along the
way.
78.                 while True:
79.                     x -= xdirection
80.                     y -= ydirection
81.                     if x == xstart and y == ystart:
```

```
82.                              break
83.                      tilesToFlip.append([x, y])
84.
85.      board[xstart][ystart] = ' ' # restore the empty space
86.      if len(tilesToFlip) == 0: # If no tiles were flipped, this is not a
valid move.
87.          return False
88.      return tilesToFlip
89.
90.
91. def isOnBoard(x, y):
92.      # Returns True if the coordinates are located on the board.
93.      return x >= 0 and x <= 7 and y >= 0 and y <=7
94.
95.
96. def getBoardWithValidMoves(board, tile):
97.      # Returns a new board with . marking the valid moves the given player
can make.
98.      dupeBoard = getBoardCopy(board)
99.
100.     for x, y in getValidMoves(dupeBoard, tile):
101.         dupeBoard[x][y] = '.'
102.     return dupeBoard
103.
104.
105. def getValidMoves(board, tile):
106.     # Returns a list of [x,y] lists of valid moves for the given player on
the given board.
107.     validMoves = []
108.
109.     for x in range(8):
110.         for y in range(8):
111.             if isValidMove(board, tile, x, y) != False:
112.                 validMoves.append([x, y])
113.     return validMoves
114.
115.
116. def getScoreOfBoard(board):
117.     # Determine the score by counting the tiles. Returns a dictionary with
keys 'X' and 'O'.
118.     xscore = 0
119.     oscore = 0
120.     for x in range(8):
121.         for y in range(8):
122.             if board[x][y] == 'X':
123.                 xscore += 1
124.             if board[x][y] == 'O':
```

```
125.                      oscore += 1
126.      return {'X':xscore, 'O':oscore}
127.
128.
129. def enterPlayerTile():
130.      # Lets the player type which tile they want to be.
131.      # Returns a list with the player's tile as the first item, and the
computer's tile as the second.
132.      tile = ''
133.      while not (tile == 'X' or tile == 'O'):
134.          print('Do you want to be X or O?')
135.          tile = input().upper()
136.
137.      # the first element in the list is the player's tile, the second is
the computer's tile.
138.      if tile == 'X':
139.          return ['X', 'O']
140.      else:
141.          return ['O', 'X']
142.
143.
144. def whoGoesFirst():
145.      # Randomly choose the player who goes first.
146.      if random.randint(0, 1) == 0:
147.          return 'computer'
148.      else:
149.          return 'player'
150.
151.
152. def playAgain():
153.      # This function returns True if the player wants to play again,
otherwise it returns False.
154.      print('Do you want to play again? (yes or no)')
155.      return input().lower().startswith('y')
156.
157.
158. def makeMove(board, tile, xstart, ystart):
159.      # Place the tile on the board at xstart, ystart, and flip any of the
opponent's pieces.
160.      # Returns False if this is an invalid move, True if it is valid.
161.      tilesToFlip = isValidMove(board, tile, xstart, ystart)
162.
163.      if tilesToFlip == False:
164.          return False
165.
166.      board[xstart][ystart] = tile
167.      for x, y in tilesToFlip:
```

```
168.              board[x][y] = tile
169.      return True
170.
171.
172. def getBoardCopy(board):
173.      # Make a duplicate of the board list and return the duplicate.
174.      dupeBoard = getNewBoard()
175.
176.      for x in range(8):
177.          for y in range(8):
178.              dupeBoard[x][y] = board[x][y]
179.
180.      return dupeBoard
181.
182.
183. def isOnCorner(x, y):
184.      # Returns True if the position is in one of the four corners.
185.      return (x == 0 and y == 0) or (x == 7 and y == 0) or (x == 0 and y ==
7) or (x == 7 and y == 7)
186.
187.
188. def getPlayerMove(board, playerTile):
189.      # Let the player type in their move.
190.      # Returns the move as [x, y] (or returns the strings 'hints' or
'quit')
191.      DIGITS1TO8 = '1 2 3 4 5 6 7 8'.split()
192.      while True:
193.          print('Enter your move, or type quit to end the game, or hints to
turn off/on hints.')
194.          move = input().lower()
195.          if move == 'quit':
196.              return 'quit'
197.          if move == 'hints':
198.              return 'hints'
199.
200.          if len(move) == 2 and move[0] in DIGITS1TO8 and move[1] in
DIGITS1TO8:
201.              x = int(move[0]) - 1
202.              y = int(move[1]) - 1
203.              if isValidMove(board, playerTile, x, y) == False:
204.                  continue
205.              else:
206.                  break
207.          else:
208.              print('That is not a valid move. Type the x digit (1-8), then
the y digit (1-8).')
209.              print('For example, 81 will be the top-right corner.')
```

```
210.
211.       return [x, y]
212.
213.
214.  def getComputerMove(board, computerTile):
215.       # Given a board and the computer's tile, determine where to
216.       # move and return that move as a [x, y] list.
217.       possibleMoves = getValidMoves(board, computerTile)
218.
219.       # randomize the order of the possible moves
220.       random.shuffle(possibleMoves)
221.
222.       # always go for a corner if available.
223.       for x, y in possibleMoves:
224.           if isOnCorner(x, y):
225.               return [x, y]
226.
227.       # Go through all the possible moves and remember the best scoring move
228.       bestScore = -1
229.       for x, y in possibleMoves:
230.           dupeBoard = getBoardCopy(board)
231.           makeMove(dupeBoard, computerTile, x, y)
232.           score = getScoreOfBoard(dupeBoard)[computerTile]
233.           if score > bestScore:
234.               bestMove = [x, y]
235.               bestScore = score
236.       return bestMove
237.
238.
239.  def showPoints(playerTile, computerTile):
240.       # Prints out the current score.
241.       scores = getScoreOfBoard(mainBoard)
242.       print('You have %s points. The computer has %s points.' %
(scores[playerTile], scores[computerTile]))
243.
244.
245.
246.  print('Welcome to Reversi!')
247.
248.  while True:
249.       # Reset the board and game.
250.       mainBoard = getNewBoard()
251.       resetBoard(mainBoard)
252.       playerTile, computerTile = enterPlayerTile()
253.       showHints = False
254.       turn = whoGoesFirst()
255.       print('The ' + turn + ' will go first.')
```

```
256.
257.     while True:
258.         if turn == 'player':
259.             # Player's turn.
260.             if showHints:
261.                 validMovesBoard = getBoardWithValidMoves(mainBoard,
playerTile)
262.                 drawBoard(validMovesBoard)
263.             else:
264.                 drawBoard(mainBoard)
265.             showPoints(playerTile, computerTile)
266.             move = getPlayerMove(mainBoard, playerTile)
267.             if move == 'quit':
268.                 print('Thanks for playing!')
269.                 sys.exit() # terminate the program
270.             elif move == 'hints':
271.                 showHints = not showHints
272.                 continue
273.             else:
274.                 makeMove(mainBoard, playerTile, move[0], move[1])
275.
276.             if getValidMoves(mainBoard, computerTile) == []:
277.                 break
278.             else:
279.                 turn = 'computer'
280.
281.         else:
282.             # Computer's turn.
283.             drawBoard(mainBoard)
284.             showPoints(playerTile, computerTile)
285.             input('Press Enter to see the computer\'s move.')
286.             x, y = getComputerMove(mainBoard, computerTile)
287.             makeMove(mainBoard, computerTile, x, y)
288.
289.             if getValidMoves(mainBoard, playerTile) == []:
290.                 break
291.             else:
292.                 turn = 'player'
293.
294.     # Display the final score.
295.     drawBoard(mainBoard)
296.     scores = getScoreOfBoard(mainBoard)
297.     print('X scored %s points. O scored %s points.' % (scores['X'],
scores['O']))
298.     if scores[playerTile] > scores[computerTile]:
299.         print('You beat the computer by %s points! Congratulations!' %
(scores[playerTile] - scores[computerTile]))
```

```
300.    elif scores[playerTile] < scores[computerTile]:
301.        print('You lost. The computer beat you by %s points.' %
(scores[computerTile] - scores[playerTile]))
302.    else:
303.        print('The game was a tie!')
304.
305.    if not playAgain():
306.        break
```

How the Code Works

The Game Board Data Structure

Before getting into the code, let's figure out the board's data structure. This data structure is a list of lists, just like the one in the previous Sonar game. The list of lists is created so that board[x][y] will represent the character on space located at position x on the X-axis (going left/right) and position y on the Y-axis (going up/down).

This character can either be a ' ' space character (to represent a blank space), a '.' period character (to represent a possible move in hint mode), or an 'X' or 'O' (to represent a player's tile). Whenever you see a parameter named board, it is meant to be this kind of list of lists data structure.

Importing Other Modules

```
1. # Reversi
2. import random
3. import sys
```

Line 2 imports the random module for its randint() and choice() functions. Line 3 imports the sys module for its exit() function.

Drawing the Board Data Structure on the Screen

```
 6. def drawBoard(board):
 7.     # This function prints out the board that it was passed. Returns None.
 8.     HLINE = '  +---+---+---+---+---+---+---+---+'
 9.     VLINE = '  |   |   |   |   |   |   |   |   |'
10.
11.     print('   1   2   3   4   5   6   7   8')
12.     print(HLINE)
```

The drawBoard() function will print the current game board based on the data structure in board. Notice that each square of the board looks like this (there could also be a '0', '.', or ' ' string instead of the 'X'):

```
+---+
|   |
| X |
|   |
+---+
```

Since the horizontal line is printed over and over again, line 8 stores it in a constant variable named HLINE. This will save you from typing out the string repeatedly.

There are also lines above and below the center of tile that are nothing but '|' characters (called "pipe" characters) with three spaces between. This is stored in a constant named VLINE.

Line 11 is the first print() function call executed, and it prints the labels for the X-axis along the top of the board. Line 12 prints the top horizontal line of the board.

```
13.     for y in range(8):
14.         print(VLINE)
15.         print(y+1, end=' ')
16.         for x in range(8):
17.             print('| %s' % (board[x][y]), end=' ')
18.         print('|')
19.         print(VLINE)
20.         print(HLINE)
```

The for loop will loop eight times, once for each row. Line 15 prints the label for the Y-axis on the left side of the board, and has an end=' ' keyword argument to print a single space instead of a new line. This is so that another loop (which again loops eight times, once for each space) prints each space (along with the 'X', '0', or ' ' character depending on what is stored in board[x][y].)

The print() function call inside the inner loop also has an end=' ' keyword argument at the end of it, meaning a space character is printed instead of a newline character. That will produce a single line on the screen that looks like '| X | X | X | X | X | X | X | X ' (if each of the board[x][y] values were 'X').

After the inner loop is done, the print() function call on line 18 prints the final '|' character along with a newline.

The code inside the outer for loop from line 14 to line 20 prints an entire row of the board like this:

```
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

When the for loop on line 13 prints the row eight times, it forms the entire board (of course, some of the spaces on the board will have '0' or ' ' instead of 'X'):

```
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
| X | X | X | X | X | X | X | X |
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

Resetting the Game Board

```
23. def resetBoard(board):
24.     # Blanks out the board it is passed, except for the original starting
position.
```

```
25.        for x in range(8):
26.            for y in range(8):
27.                board[x][y] = ' '
```

Line 25 and 26 have nested loops to set the board data structure to be all single-space strings. This makes a blank Reversi board. The resetBoard() function is called as part of starting a new game.

Setting Up the Starting Pieces

```
29.        # Starting pieces:
30.        board[3][3] = 'X'
31.        board[3][4] = 'O'
32.        board[4][3] = 'O'
33.        board[4][4] = 'X'
```

At the beginning of a game, each player has two tiles already laid down in the center. Lines 30 to 33 set those tiles on the blank board.

The resetBoard() function does not have to return the board variable, because board is a reference to a list. Making changes inside the function's local scope will modify the original list that was passed as the argument. (See the References section in Chapter 10.)

Creating a New Game Board Data Structure

```
36. def getNewBoard():
37.     # Creates a brand new, blank board data structure.
38.     board = []
39.     for i in range(8):
40.         board.append([' '] * 8)
41.
42.     return board
```

The getNewBoard() function creates a new board data structure and returns it. Line 38 creates the outer list and stores a reference to this list in board. Line 40 creates the inner lists using list replication. ([' '] * 8 evaluates to be the same as [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '] but with less typing.)

Line 39's for loop here creates the eight inner lists. The spaces represent a completely empty game board.

What board ends up being is a list of eight lists, and each of those eight lists themselves has eight strings. The result is 64 ' ' strings.

Checking if a Move is Valid

```
45. def isValidMove(board, tile, xstart, ystart):
46.     # Returns False if the player's move on space xstart, ystart is
invalid.
47.     # If it is a valid move, returns a list of spaces that would become
the player's if they made a move here.
48.     if board[xstart][ystart] != ' ' or not isOnBoard(xstart, ystart):
49.         return False
50.     board[xstart][ystart] = tile # temporarily set the tile on the board.
51.     if tile == 'X':
52.         otherTile = 'O'
53.     else:
54.         otherTile = 'X'
55.     tilesToFlip = []
```

Given a board data structure, the player's tile, and the XY coordinates for player's move, isValidMove() should return True if the Reversi game rules allow a move to those coordinates and False if they don't.

Line 48 checks if the XY coordinates are not on the game board, or if the space isn't empty. isOnBoard() is a function defined later in the program that makes sure both the X and Y coordinates are between 0 and 7.

The next step is to temporarily place the player's tile on the board. This tile will be removed (by setting the board space back to ' ' before returning).

The player's tile (either the human player or the computer player) is in tile, but this function will need to know the other player's tile. If the player's tile is 'X' then obviously the other player's tile is 'O', and vice versa.

Finally, if the given XY coordinate ends up as a valid position, isValidMove() returns a list of all the opponent's tiles that would be flipped by this move.

```
59.     for xdirection, ydirection in [[0, 1], [1, 1], [1, 0], [1, -1], [0, -
1], [-1, -1], [-1, 0], [-1, 1]]:
```

The for loop iterates through a list of lists which represent directions you can move on the game board. The game board is a Cartesian coordinate system with an X and Y direction. There are eight directions you can move: up, down, left, right, and the four diagonal directions. Each of the eight 2-item lists in the list on line 59 is used for moving in one of these directions. The program moves in a direction by adding the first value in the two-item list to the X coordinate, and the second value to the Y coordinate.

Because the X coordinates increase as you go to the right, you can "move" to the right by adding
1 to the X coordinate. So the [1, 0] list adds 1 to the X coordinate and 0 to the Y coordinate,
resulting in "movement" to the right. Moving to the left is the opposite: you would subtract 1
(that is, add -1) from the X coordinate.

But to move diagonally, you need to add or subtract to both coordinates. For example, adding 1 to
the X coordinate to move right and adding -1 to the Y coordinate to move up would result in
moving to the up-right diagonal direction.

Checking Each of the Eight Directions

Here is a diagram to make it easier to remember which two-item list represents which direction:
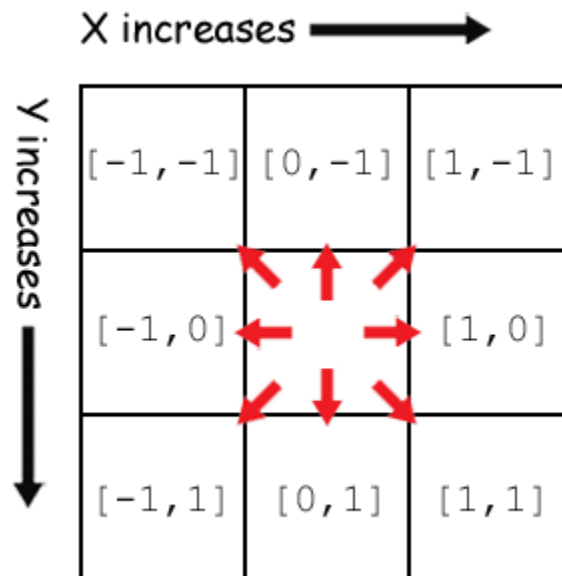

Figure 15-7: Each two-item list represents one of the eight directions.

```
59.       for xdirection, ydirection in [[0, 1], [1, 1], [1, 0], [1, -1], [0, -
1], [-1, -1], [-1, 0], [-1, 1]]:
60.           x, y = xstart, ystart
61.           x += xdirection # first step in the direction
62.           y += ydirection # first step in the direction
```

Line 60 sets an x and y variable to be the same value as xstart and ystart, respectively, using
multiple assignment. Change x and y to "move" in the direction that xdirection and
ydirection dictate. The xstart and ystart variables will stay the same so that the program can
remember which space it originally started from.

```
63.             if isOnBoard(x, y) and board[x][y] == otherTile:
64.                 # There is a piece belonging to the other player next to our
piece.
65.                 x += xdirection
66.                 y += ydirection
67.                 if not isOnBoard(x, y):
68.                     continue
```

Remember, in order for this to be a valid move, the first step in this direction must be 1) on the board and 2) must be occupied by the other player's tile. Otherwise there aren't any of the opponent's tiles to flip, and a valid move must flip over at least one tile. If these two things aren't true, line 63's condition isn't `True` and the execution goes back to the `for` statement for the next direction.

But if the first space does have the other player's tile, then the program should keep checking in that direction until it reaches one of the player's tiles. If it reaches past the end of the board though, then line 68 should continue back to the `for` statement to try the next direction.

```
69.             while board[x][y] == otherTile:
70.                 x += xdirection
71.                 y += ydirection
72.                 if not isOnBoard(x, y): # break out of while loop, then
continue in for loop
73.                     break
74.             if not isOnBoard(x, y):
75.                 continue
```

The `while` loop on line 69 keeps looping so that x and y keep going in the current direction as long as it keeps seeing a trail of the other player's tiles. If line 72 detects that x and y moved off of the board, line 73 breaks out of the `for` loop and the flow of execution moves to line 74.

What you really want to do is break out of the `while` loop but continue in the `for` loop. This is why line 74 rechecks `not isOnBoard(x, y)` and runs `continue`, which moves execution to the next direction in line 59's `for` statement. Remember, `break` and `continue` statements will only break or continue from the innermost loop they are in.

Finding Out if There are Pieces to Flip Over

```
76.             if board[x][y] == tile:
77.                 # There are pieces to flip over. Go in the reverse
direction until we reach the original space, noting all the tiles along the
way.
78.                 while True:
```

```
79.                         x -= xdirection
80.                         y -= ydirection
81.                         if x == xstart and y == ystart:
82.                             break
83.                         tilesToFlip.append([x, y])
```

Line 69's while loop stops looping when the code has reached the end of the otherTile tiles.
Line 76 checks if this space on the board holds one of our tiles. If it does, then the move
originally passed to isValidMove() is valid.

Line 78 loops by moving x and y in reverse back to the original xstart and ystart position by
subtracting x and y. Each space is appended to the tilesToFlip list.

```
85.     board[xstart][ystart] = ' ' # restore the empty space
86.     if len(tilesToFlip) == 0: # If no tiles were flipped, this is not a
valid move.
87.         return False
88.     return tilesToFlip
```

The for loop that started on line 59 does this in all eight directions. After that loop is done, the
tilesToFlip list will contain the XY coordinates all of our opponent's tiles that would be flipped
if the player moved on xstart, ystart. Remember, the isValidMove() function is only
checking to see if the original move was valid. It doesn't actually permanently change the data
structure of the game board.

If none of the eight directions ended up flipping at least one of the opponent's tiles, then
tilesToFlip will be an empty list. This is a sign that this move is not valid and isValidMove()
should return False.

Otherwise, isValidMove() returns tilesToFlip.

Checking for Valid Coordinates

```
91. def isOnBoard(x, y):
92.     # Returns True if the coordinates are located on the board.
93.     return x >= 0 and x <= 7 and y >= 0 and y <=7
```

isOnBoard() is a function called from isValidMove(). Calling the function is shorthand for the
Boolean expression on line 93 that is True if both x and y are between 0 and 7. This function

checks if an X and Y coordinate is actually on the game board. For example, an X coordinate of 4 and a Y coordinate of 9999 would not be on the board since Y coordinates only go up to 7.

Getting a List with All Valid Moves

```
 96. def getBoardWithValidMoves(board, tile):
 97.     # Returns a new board with . marking the valid moves the given player
can make.
 98.     dupeBoard = getBoardCopy(board)
 99.
100.     for x, y in getValidMoves(dupeBoard, tile):
101.         dupeBoard[x][y] = '.'
102.     return dupeBoard
```

getBoardWithValidMoves() returns a game board data structure that has '.' characters for all spaces that are valid moves. The periods are for the hints mode that displays a board with all possible moves marked on it.

This function creates a duplicate game board data structure (returned by getBoardCopy() on line 98) instead of modifying the one passed to it in the board parameter. Line 100 calls getValidMoves() to get a list of XY coordinates with all the legal moves the player could make. The board copy is marked with periods in those spaces and returned.

```
105. def getValidMoves(board, tile):
106.     # Returns a list of [x,y] lists of valid moves for the given player on
the given board.
107.     validMoves = []
108.
109.     for x in range(8):
110.         for y in range(8):
111.             if isValidMove(board, tile, x, y) != False:
112.                 validMoves.append([x, y])
113.     return validMoves
```

The getValidMoves() function returns a list of two-item lists. These lists hold the XY coordinates for all valid moves for tile's player for the board data structure in the board parameter.

This function uses nested loops (on lines 109 and 110) to check every XY coordinate (all sixty four of them) by calling isValidMove() on that space and checking if it returns False or a list of possible moves (in which case it is a valid move). Each valid XY coordinate is appended to the list in validMoves.

The `bool()` Function

The `bool()` is similar to the `int()` and `str()` functions. It returns the Boolean value form of the value passed to it.

Most data types have one value that is considered the `False` value for that data type. Every other value is consider `True`. For example, the integer `0`, the floating point number `0.0`, the empty string, the empty list, and the empty dictionary are all considered to be `False` when used as the condition for an `if` or loop statement. All other values are `True`. Try entering the following into the interactive shell:

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
>>> bool({})
False
>>> bool(1)
True
>>> bool('Hello')
True
>>> bool([1, 2, 3, 4, 5])
True
>>> bool({'spam':'cheese', 'fizz':'buzz'})
True
```

Imagine that any statement's condition is placed inside a call to `bool()`. That is, conditions are automatically interpreted as Boolean values. This is why the condition on line 111 works correctly. The call to the `isValidMove()` function either returns the Boolean value `False` or a non-empty list.

If you imagine that the entire condition is placed inside a call to `bool()`, then line 111's condition `False` becomes `bool(False)` (which, of course, evaluates to `False`). And a condition of a non-empty list placed as the parameter to `bool()` will return `True`.

Getting the Score of the Game Board

```
116. def getScoreOfBoard(board):
117.     # Determine the score by counting the tiles. Returns a dictionary with
keys 'X' and 'O'.
118.     xscore = 0
```

```
119.        oscore = 0
120.        for x in range(8):
121.            for y in range(8):
122.                if board[x][y] == 'X':
123.                    xscore += 1
124.                if board[x][y] == 'O':
125.                    oscore += 1
126.        return {'X':xscore, 'O':oscore}
```

The getScoreOfBoard() function uses nested for loops to check all 64 spaces on the board (8 rows times 8 columns per row is 64 spaces) and see which tile (if any) is on them. For each 'X' tile, the code increments xscore on line 123. For each 'O' tile, the code increments oscore on line 125

Getting the Player's Tile Choice

```
129. def enterPlayerTile():
130.     # Lets the player type which tile they want to be.
131.     # Returns a list with the player's tile as the first item, and the
computer's tile as the second.
132.     tile = ''
133.     while not (tile == 'X' or tile == 'O'):
134.         print('Do you want to be X or O?')
135.         tile = input().upper()
```

This function asks the player which tile they want to be, either 'X' or 'O'. The for loop will keep looping until the player types in 'X' or 'O'.

```
137.     # the first element in the list is the player's tile, the second is
the computer's tile.
138.     if tile == 'X':
139.         return ['X', 'O']
140.     else:
141.         return ['O', 'X']
```

The enterPlayerTile() function then returns a two-item list, where the player's tile choice is the first item and the computer's tile is the second. Line 252, which calls enterPlayerTile(), uses multiple assignment to put these two returned items in two variables.

Determining Who Goes First

```
144. def whoGoesFirst():
145.     # Randomly choose the player who goes first.
```

```
146.        if random.randint(0, 1) == 0:
147.            return 'computer'
148.        else:
149.            return 'player'
```

The `whoGoesFirst()` function randomly selects who goes first, and returns either the string `'computer'` or the string `'player'`.

Asking the Player to Play Again

```
152. def playAgain():
153.     # This function returns True if the player wants to play again,
otherwise it returns False.
154.     print('Do you want to play again? (yes or no)')
155.     return input().lower().startswith('y')
```

The `playAgain()` function was also in previous games. If the player types in a string that begins with `'y'`, then the function returns `True`. Otherwise the function returns `False`.

Placing Down a Tile on the Game Board

```
158. def makeMove(board, tile, xstart, ystart):
159.     # Place the tile on the board at xstart, ystart, and flip any of the
opponent's pieces.
160.     # Returns False if this is an invalid move, True if it is valid.
161.     tilesToFlip = isValidMove(board, tile, xstart, ystart)
```

`makeMove()` is called when you want to place a tile on the board and flip the other tiles according to the rules of Reversi. This function modifies the `board` data structure that is passed in-place. Changes made to the `board` variable (because it is a list reference) will be made to the global scope.

Most of the work is done by `isValidMove()`, which returns a list of XY coordinates (in a two-item list) of tiles that need to be flipped. (Remember, if the `xstart` and `ystart` arguments point to an invalid move, then `isValidMove()` will return the Boolean value `False`.)

```
163.     if tilesToFlip == False:
164.         return False
165.
166.     board[xstart][ystart] = tile
167.     for x, y in tilesToFlip:
168.         board[x][y] = tile
```

```
169.        return True
```

On lines 163 and 164, if the return value of isValidMove() (now stored in tilesToFlip) was False, then makeMove() will also return False.

Otherwise, isValidMove() returns a list of spaces on the board to put down the tiles (the 'X' or 'O' string in tile). Line 166 sets the space that the player has moved on. Line 167's for loop sets all the tiles that are in tilesToFlip.

Copying the Board Data Structure

```
172. def getBoardCopy(board):
173.     # Make a duplicate of the board list and return the duplicate.
174.     dupeBoard = getNewBoard()
175.
176.     for x in range(8):
177.         for y in range(8):
178.             dupeBoard[x][y] = board[x][y]
179.
180.     return dupeBoard
```

getBoardCopy() is different from getNewBoard(). getNewBoad() will create a blank game board data structure which has only empty spaces and the four starting tiles. getBoardCopy() will create a blank game board data structure, but then copy all of the spaces from the board parameter. This function is used by the AI to have a game board that it can change around without changing the real game board. This technique was also used by the previous Tic Tac Toe program.

A call to getNewBoard() handles getting a fresh game board data structure. Then the two nested for loops copy each of the 64 tiles from board to the duplicate board data structure in dupeBoard.

Determining if a Space is on a Corner

```
183. def isOnCorner(x, y):
184.     # Returns True if the position is in one of the four corners.
185.     return (x == 0 and y == 0) or (x == 7 and y == 0) or (x == 0 and y ==
7) or (x == 7 and y == 7)
```

The isOnCorner() function returns True if the coordinates are on a corner space at coordinates (0,0), (7,0), (0,7) or (7,7). Otherwise isOnCorner() returns False.

Getting the Player's Move

```
188. def getPlayerMove(board, playerTile):
189.     # Let the player type in their move.
190.     # Returns the move as [x, y] (or returns the strings 'hints' or
'quit')
191.     DIGITS1TO8 = '1 2 3 4 5 6 7 8'.split()
```

The getPlayerMove() function is called to let the player type in the coordinates of their next move (and check if the move is valid). The player can also type in 'hints' to turn hints mode on (if it is off) or off (if it is on). The player can also type in 'quit' to quit the game.

The DIGITS1TO8 constant variable is the list ['1', '2', '3', '4', '5', '6', '7', '8']. The DIGITS1TO8 constant is used because it is easier to type than the entire list. You can't use the isdigit() method because that would allow 0 and 9 to be entered, which are not valid coordinates on the 8×8 board.

```
192.     while True:
193.         print('Enter your move, or type quit to end the game, or hints to
turn off/on hints.')
194.         move = input().lower()
195.         if move == 'quit':
196.             return 'quit'
197.         if move == 'hints':
198.             return 'hints'
```

The while loop will keep looping until the player has typed in a valid move. Lines 195 to 198 check if the player wants to quit or toggle hints mode, and return the string 'quit' or 'hints', respectively. The lower() method is called on the string returned by input() so the player can type 'HINTS' or 'Quit' but still have the command understood.

The code that called getPlayerMove() will handle what to do if the player wants to quit or toggle hints mode.

```
200.         if len(move) == 2 and move[0] in DIGITS1TO8 and move[1] in
DIGITS1TO8:
201.             x = int(move[0]) - 1
202.             y = int(move[1]) - 1
203.             if isValidMove(board, playerTile, x, y) == False:
204.                 continue
205.             else:
206.                 break
```

The game is expecting that the player would have typed in the XY coordinates of their move as two numbers without anything between them. Line 200 first checks that the size of the string the player typed in is 2. After that, it also checks that both move[0] (the first character in the string) and move[1] (the second character in the string) are strings that exist in DIGITS1TO8.

Remember that the game board data structures have indexes from 0 to 7, not 1 to 8. The code prints 1 to 8 when the board is displayed in drawBoard() because non-programmers are used to numbers beginning at 1 instead of 0. So to convert the strings in move[0] and move[1] to integers, lines 201 and 202 subtract 1.

Even if the player typed in a correct move, the code still needs to check that the move is allowed by the rules of Reversi. This is done by isValidMove() which is passed the game board data structure, the player's tile, and the XY coordinates of the move.

If isValidMove() returns False, then line 204's continue statement executes. The execution will then go back to the beginning of the while loop and asks the player for a valid move again.

Otherwise, the player did type in a valid move and the execution needs to break out of the while loop.

```
207.          else:
208.               print('That is not a valid move. Type the x digit (1-8), then
the y digit (1-8).')
209.               print('For example, 81 will be the top-right corner.')
```

If the if statement's condition on line 200 was False, then the player didn't type in a valid move. Lines 208 and 209 instructs them on how to correctly type in moves. Afterwards, the execution moves back to the while statement on line 192 because line 209 isn't only the last line in the else-block, but also the last line in the while-block.

```
211.     return [x, y]
```

Finally, getPlayerMove() returns a two-item list with the XY coordinates of the player's valid move.

Getting the Computer's Move

```
214. def getComputerMove(board, computerTile):
215.     # Given a board and the computer's tile, determine where to
216.     # move and return that move as a [x, y] list.
217.     possibleMoves = getValidMoves(board, computerTile)
```

getComputerMove() and is where the AI algorithm is implemented. Normally you use the results from getValidMoves() for hints mode. Hints mode will print '.' period characters on the board to show the player all the potential moves they can make.

But if getValidMoves() is called with the computer AI's tile (in computerTile), it will also find all the possible moves that the computer can make. The AI will select the best move from this list.

```
219.        # randomize the order of the possible moves
220.        random.shuffle(possibleMoves)
```

First, random.shuffle() will randomize the order of moves in the possibleMoves list. Why we want to shuffle the possibleMoves list will be explained later, but first let's look at the algorithm.

Corner Moves are the Best Moves

```
222.        # always go for a corner if available.
223.        for x, y in possibleMoves:
224.            if isOnCorner(x, y):
225.                return [x, y]
```

First, line 223 loops through every move in possibleMoves. If any of them are on the corner, return that space is returned as the move. Corner moves are a good idea in Reversi because once a tile has been placed on the corner, it can never be flipped over. Since possibleMoves is a list of two-item lists, use multiple assignment in the for loop to set x and y.

If possibleMoves contains multiple corner moves, the first one is always used. But since possibleMoves was shuffled on line 220, it is random which corner move is first in the list.

Get a List of the Best Scoring Moves

```
227.        # Go through all the possible moves and remember the best scoring move
228.        bestScore = -1
229.        for x, y in possibleMoves:
230.            dupeBoard = getBoardCopy(board)
231.            makeMove(dupeBoard, computerTile, x, y)
232.            score = getScoreOfBoard(dupeBoard)[computerTile]
233.            if score > bestScore:
234.                bestMove = [x, y]
235.                bestScore = score
236.        return bestMove
```

If there are no corner moves, loop through the entire list and find out which move results in the highest score. Line 229's for loop will set x and y to every move in possibleMoves. bestMove is

set to the highest scoring move the code has found so far, and `bestScore` is set to the best move's score.

When the code in the loop finds a move that scores higher than `bestScore`, line 233 to 235 will store that move and score as the new values in `bestMove` and `bestScore`.

Simulate All Possible Moves on Duplicate Board Data Structures

Before simulating a move, line 230 makes a duplicate game board data structure by calling `getBoardCopy()`. You'll want a copy so you can modify without changing the real game board data structure stored in the `board` variable.

Then line 231 calls `makeMove()`, passing the duplicate board (stored in `dupeBoard`) instead of the real board. This will simulate what would happen on the real board if this move was made. `makeMove()` will handle placing the computer's tile and the flipping the player's tiles on the duplicate board.

Line 232 calls `getScoreOfBoard()`with the duplicate board, which returns a dictionary where the keys are `'X'` and `'O'`, and the values are the scores.

For example, pretend that `getScoreOfBoard()` returns the dictionary `{'X':22, 'O':8}` and `computerTile` is `'X'`. Then `getScoreOfBoard(dupeBoard)[computerTile]` would evaluate to `{'X':22, 'O':8}['X']`, which would then evaluate to 22. If 22 is larger than `bestScore`, `bestScore` is set to `22` and `bestMove` is set to the current x and y values.

By the time this `for` loop is finished, you can be sure that `bestScore` is the highest possible score a move can make, and that move is stored in `bestMove`.

Line 228 first sets `bestScore` to `-1` so that the first move the code checks will be set to the first `bestMove`. This will guarantee that `bestMove` is set to one of the moves from `possibleMoves` when it returns.

Even though the code always chooses the first in the list of these tied moves, it's random because the list order was shuffled on line 220. This ensures that the AI won't be predictable when there's more than one best move.

Printing the Scores to the Screen

```
239. def showPoints(playerTile, computerTile):
240.     # Prints out the current score.
241.     scores = getScoreOfBoard(mainBoard)
242.     print('You have %s points. The computer has %s points.' %
(scores[playerTile], scores[computerTile]))
```

showPoints() calls the getScoreOfBoard() function and then prints the player's and computer's scores. Remember that getScoreOfBoard() returns a dictionary with the keys 'X' and 'O' and values of the scores for the X and O players.

That's all the functions for the Reversi game. The code starting on line 246 will implement the actual game and calls these functions as needed.

The Start of the Game

```
246. print('Welcome to Reversi!')
247.
248. while True:
249.     # Reset the board and game.
250.     mainBoard = getNewBoard()
251.     resetBoard(mainBoard)
252.     playerTile, computerTile = enterPlayerTile()
253.     showHints = False
254.     turn = whoGoesFirst()
255.     print('The ' + turn + ' will go first.')
```

The while loop on line 248 is the main game loop. The program will loop back to line 248 when a new game starts. First get a new game board data structure by calling getNewBoard() and set the starting tiles by calling resetBoard(). mainBoard is the main game board data structure for the program. The call to enterPlayerTile() will let the player type in whether they want to be 'X' or 'O'. The return value is then stored in playerTile and computerTile using multiple assignment.

showHints is a Boolean value that determines if hints mode is on or off. It starts as off as False on line 253.

The turn variable is a string that either has the string value 'player' or 'computer'. It will keep track of whose turn it is. It is set to the return value of whoGoesFirst(), which randomly chooses who will go first.

Running the Player's Turn

```
257.     while True:
258.         if turn == 'player':
259.             # Player's turn.
260.             if showHints:
261.                 validMovesBoard = getBoardWithValidMoves(mainBoard,
playerTile)
262.                 drawBoard(validMovesBoard)
263.             else:
```

```
264.                    drawBoard(mainBoard)
265.                showPoints(playerTile, computerTile)
```

The while loop that starts on line 257 will keep looping each time the player or computer takes a turn. The execution will break out of this loop when the current game is over.

Line 258 has an if statement whose body has the code that runs if it is the player's turn. (The else-block that starts on line 282 has the code for the computer's turn.)

First the board is displayed on the screen. If hints mode is on (that is, showHints is True), then the board data structure needs to have '.' period characters on every valid space the player could move.

The getBoardWithValidMoves() function does that. It is passed a game board data structure and returns a copy that also contains '.' period characters. Line 262 passes this board to the drawBoard() function.

If hints mode is off, then line 264 passes mainBoard to drawBoard().

After printing out the game board to the player, you also want to print the current score by calling showPoints() on line 265.

```
266.                move = getPlayerMove(mainBoard, playerTile)
```

Next, let the player type in their move. getPlayerMove() handles this, and its return value is a two-item list of the XY coordinate of the player's move. getPlayerMove() has already made sure that the move the player typed in is a valid move.

Handling the Quit or Hints Commands

```
267.                if move == 'quit':
268.                    print('Thanks for playing!')
269.                    sys.exit() # terminate the program
270.                elif move == 'hints':
271.                    showHints = not showHints
272.                    continue
273.                else:
274.                    makeMove(mainBoard, playerTile, move[0], move[1])
```

If the player typed in the string 'quit' for their move, then getPlayerMove() would have returned the string 'quit'. In that case, line 269 calls the sys.exit() to terminate the program.

If the player typed in the string `'hints'` for their move, then `getPlayerMove()` would have returned the string `'hints'`. In that case, you want to turn hints mode on (if it was off) or off (if it was on).

The `showHints = not showHints` assignment statement on line 271 handles both of these cases, because `not False` evaluates to `True` and `not True` evaluates to `False`. Then the `continue` statement moves the execution to the start of the loop (`turn` has not changed, so it will still be the player's turn).

Otherwise, if the player didn't quit or toggle hints mode, line 274 calls `makeMove()` to make the player's move on the board.

Make the Player's Move

```
276.            if getValidMoves(mainBoard, computerTile) == []:
277.                break
278.            else:
279.                turn = 'computer'
```

After making the player's move, line 276 calls `getValidMoves()` to see if the computer could make any moves. If `getValidMoves()` returns a blank list, then there are no more valid moves that the computer could make. In that case, line 277 breaks out of the `while` loop and ends the game.

Otherwise, line 279 sets `turn` to `'computer'`. The flow of execution skips the else-block and reaches the end of the while-block, so execution jumps back to the `while` statement on line 257. This time, however, it will be the computer's turn.

Running the Computer's Turn

```
281.        else:
282.            # Computer's turn.
283.            drawBoard(mainBoard)
284.            showPoints(playerTile, computerTile)
285.            input('Press Enter to see the computer\'s move.')
286.            x, y = getComputerMove(mainBoard, computerTile)
287.            makeMove(mainBoard, computerTile, x, y)
```

After printing out the board with `drawBoard()`, also print the current score with a call to `showPoints()` on line 284.

Line 285 calls `input()` to pause the script while the player can look at the board. This is much like how `input()` was used to pause the program in the Jokes chapter. Instead of using a `print()`

call to print a string before a call to `input()`, you can do the same thing by passing the string to print to `input()`.

After the player has looked at the board and pressed ENTER, line 286 calls `getComputerMove()` to get the XY coordinates of the computer's next move. These coordinates are stored in variables `x` and `y` using multiple assignment.

Finally, pass `x` and `y`, along with the game board data structure and the computer's tile, to the `makeMove()` function. This places the computer's tile on the game board in `mainBoard` to reflect the computer's move. Line 286's call to `getComputerMove()` got the computer's move (and stored it in variables `x` and `y`). The call to `makeMove()` on line 287 makes the move on the board.

```
289.                 if getValidMoves(mainBoard, playerTile) == []:
290.                     break
291.                 else:
292.                     turn = 'player'
```

Lines 289 to 292 are similar to lines 276 to 279. After the computer has made its move, line 289 checks if there exist any valid moves the human player can make. If `getValidMoves()` returns an empty list, then there are no valid moves. That means the game is over, and line 290 breaks out of the `while` loop.

Otherwise, there's at least one possible move the player should make. The `turn` variable is to `'player'`. There is no more code in the while-block after line 292, so execution loops back to the `while` statement on line 257.

Drawing Everything on the Screen

```
294.     # Display the final score.
295.     drawBoard(mainBoard)
296.     scores = getScoreOfBoard(mainBoard)
297.     print('X scored %s points. O scored %s points.' % (scores['X'],
scores['O']))
298.     if scores[playerTile] > scores[computerTile]:
299.         print('You beat the computer by %s points! Congratulations!' %
(scores[playerTile] - scores[computerTile]))
300.     elif scores[playerTile] < scores[computerTile]:
301.         print('You lost. The computer beat you by %s points.' %
(scores[computerTile] - scores[playerTile]))
302.     else:
303.         print('The game was a tie!')
```

Line 294 is the first line beyond the while-block that started on line 257. This code is executed when the execution breaks out of that `while` loop from line 290 or 277. At this point, the game is over. Now the program should print the board and scores and determine who won the game.

`getScoreOfBoard()` will return a dictionary with keys `'X'` and `'O'` and values of both players' scores. By checking if the player's score is greater than, less than, or equal to the computer's score, you can know if the player won, lost, or tied, respectively.

Ask the Player to Play Again

```
305.        if not playAgain():
306.            break
```

Call the `playAgain()` function, which returns `True` if the player typed in that they want to play another game. If `playAgain()` returns `False`, the `not` operator makes the `if` statement's condition `True`, the execution breaks out of the `while` loop that started on line 248. Since there are no more lines of code after this while-block, the program terminates.

Otherwise, `playAgain()` has returned `True` (making the `if` statement's condition `False`), and so execution loops back to the `while` statement on line 248 and a new game board is created.

Changing the `drawBoard()` Function

The board you draw for the Reversi game is large. But you could change the `drawBoard()` function's code to draw out a much smaller board, while keeping the rest of the game code the same. The new, smaller board would look like this:

```
   12345678
 +--------+
1|   O    |
2|   XOX  |
3|   O    |
4| XXXXX  |
5|  .OX   |
6|  OOO   |
7| ..O..  |
8|   O    |
 +--------+
You have 8 points. The computer has 9 points.
Enter your move, or type quit to end the game, or hints to turn off/on hints.
```

Here is the code for this new `drawBoard()` function, starting at line 6. You can also download this code from http://invpy.com/reversi_mini.py.

```
 6. def drawBoard(board):
 7.     # This function prints out the board that it was passed. Returns None.
 8.     HLINE = '  +--------+'
 9.     print('  12345678')
10.     print(HLINE)
11.     for y in range(8):
12.         print('%s|' % (y+1), end='')
13.         for x in range(8):
14.             print(board[x][y], end='')
15.         print('|')
16.     print(HLINE)
```

Summary

The AI may seem almost unbeatable, but this isn't because the computer is smart. The strategy it follows is simple: move on the corner if you can, otherwise make the move that will flip over the most tiles. We could do that, but it would be slow to figure out how many tiles would be flipped for every possible valid move we could make. But calculating this for the computer is simple. The computer isn't smarter than us, it's just much faster!

This game is similar to Sonar because it makes use of a grid for a board. It is also like the Tic Tac Toe game because there's an AI that plans out the best move for it to take. This chapter only introduced one new concept: that empty lists, blank strings, and the integer 0 all evaluate to False in the context of a condition.

Other than that, this game used programming concepts you already knew! You don't have to know much about programming to create interesting games. However, this game is stretching how far you can get with ASCII art. The board took up almost the entire screen to draw, and the game didn't have any color.

Later in this book, we will learn how to create games with graphics and animation, not just text. We will do this using a module called Pygame, which adds new functions and features to Python so that we can break away from using only text and keyboard input.

Chapter 16

REVERSI AI SIMULATION

> Topics Covered In This Chapter:
> - Simulations
> - Percentages
> - Pie Charts
> - Integer Division
> - The round() function
> - "Computer vs. Computer" Games

The Reversi AI algorithm was simple, but it beats me almost every time I play it. This is because the computer can process instructions fast, so checking each possible position on the board and selecting the highest scoring move is easy for the computer. It would take a long time for me to find the best move this way.

The Reversi program in Chapter 14 had two functions, getPlayerMove() and getComputerMove() which both returned the move selected as a two-item list like [x, y]. The both also had the same parameters, the game board data structure and which tile they were. getPlayerMove() decided which [x, y] move to return by letting the player type in the coordinates. getComputerMove() decided which [x, y] move to return by running the Reversi AI algorithm.

What happens when we replace the call to getPlayerMove() with a call to getComputerMove()? Then the player never types in a move, it is decided for them! The computer is playing against itself!

We will make three new programs, each based on the Reversi program in the last chapter:

- *AISim1.py* will be made by making changes to *reversi.py*
- *AISim2.py* will be made by making changes to *AISim1.py*
- *AISim3.py* will be made by making changes to *AISim2.py*

You can either type these changes in yourself, or download them from the book's website at the URL http://invpy.com/chap16.

Making the Computer Play Against Itself

Save the old *reversi.py* file as *AISim1.py* by clicking on **File ▶ Save As**, and then entering *AISim1.py* for the file name and clicking Ok. This will create a copy of our Reversi source code as a new file that you can make changes to, while leaving the original Reversi game the same (you may want to play it again). Change the following code in *AISim1.py*:

```
266. move = getPlayerMove(mainBoard, playerTile)
```

To this (the change is in **bold**):

```
266. move = getComputerMove(mainBoard, playerTile)
```

Now run the program. Notice that the game still asks you if you want to be X or O, but it won't ask you to enter any moves. When you replaced `getPlayerMove()`, you no longer call any code that takes this input from the player. You still press ENTER after the original computer's moves (because of the `input('Press Enter to see the computer\'s move.')` on line 285), but the game plays itself!

Let's make some other changes to *AISim1.py*. All of the functions you defined for Reversi can stay the same. But replace the entire main section of the program (line 246 and on) to look like the following code. Some of the code has remained, but most of it has been altered. But all of the lines before line 246 are the same as in Reversi in the last chapter. You can also avoid typing in the code by downloading the source from the URL http://invpy.com/chap16.

If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/AISim1.

```
                                                        AISim1.py
246. print('Welcome to Reversi!')
247.
248. while True:
249.     # Reset the board and game.
250.     mainBoard = getNewBoard()
251.     resetBoard(mainBoard)
252.     if whoGoesFirst() == 'player':
253.         turn = 'X'
254.     else:
255.         turn = 'O'
256.     print('The ' + turn + ' will go first.')
257.
258.     while True:
259.         drawBoard(mainBoard)
```

```
260.            scores = getScoreOfBoard(mainBoard)
261.            print('X has %s points. O has %s points' % (scores['X'],
scores['O']))
262.            input('Press Enter to continue.')
263.
264.            if turn == 'X':
265.                # X's turn.
266.                otherTile = 'O'
267.                x, y = getComputerMove(mainBoard, 'X')
268.                makeMove(mainBoard, 'X', x, y)
269.            else:
270.                # O's turn.
271.                otherTile = 'X'
272.                x, y = getComputerMove(mainBoard, 'O')
273.                makeMove(mainBoard, 'O', x, y)
274.
275.            if getValidMoves(mainBoard, otherTile) == []:
276.                break
277.            else:
278.                turn = otherTile
279.
280.        # Display the final score.
281.        drawBoard(mainBoard)
282.        scores = getScoreOfBoard(mainBoard)
283.        print('X scored %s points. O scored %s points.' % (scores['X'],
scores['O']))
284.
285.        if not playAgain():
286.            sys.exit()
```

How the AISim1.py Code Works

The *AISim1.py* program is the same as the original Reversi program, except that the call to getPlayerMove() has been replaced with a call to getComputerMove(). There have been some other changes to the text that is printed to the screen to make the game easier to follow.

When you run the *AISim1.py* program, all you can do is press Enter for each turn until the game ends. Run through a few games and watch the computer play itself. Since both the X and O players are using the same algorithm, it really is just a matter of luck to see who wins. The X player will win half the time, and the O player will win half the time.

Making the Computer Play Itself Several Times

But what if we created a new algorithm? Then we could set this new AI against the one implemented in getComputerMove(), and see which one is better. Let's make some changes to the source code. Do the following to make *AISim2.py*:

1. Click on **File ► Save As**.
2. Save this file as *AISim2.py* so that you can make changes without affecting *AISim1.py*. (At this point, *AISim1.py* and *AISim2.py* will have the same code.)
3. Make changes to *AISim2.py* and save that file. (*AISim2.py* will have the new changes and *AISim1.py* will have the original, unchanged code.)

Add the following code. The additions are in bold, and some lines have been removed. When you are done changing the file, save it as *AISim2.py*.

If this is confusing, you can always download the *AISim2.py* source code from the book's website at http://invpy.com/chap16.

AISim2.py

If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/AISim2.

```
                                                              AISim2.py
246. print('Welcome to Reversi!')
247.
248. xwins = 0
249. owins = 0
250. ties = 0
251. numGames = int(input('Enter number of games to run: '))
252.
253. for game in range(numGames):
254.     print('Game #%s:' % (game), end=' ')
255.     # Reset the board and game.
256.     mainBoard = getNewBoard()
257.     resetBoard(mainBoard)
258.     if whoGoesFirst() == 'player':
259.         turn = 'X'
260.     else:
261.         turn = 'O'
262.
263.     while True:
264.         if turn == 'X':
265.             # X's turn.
```

```
266.                otherTile = 'O'
267.                x, y = getComputerMove(mainBoard, 'X')
268.                makeMove(mainBoard, 'X', x, y)
269.            else:
270.                # O's turn.
271.                otherTile = 'X'
272.                x, y = getComputerMove(mainBoard, 'O')
273.                makeMove(mainBoard, 'O', x, y)
274.
275.            if getValidMoves(mainBoard, otherTile) == []:
276.                break
277.            else:
278.                turn = otherTile
279.
280.        # Display the final score.
281.        scores = getScoreOfBoard(mainBoard)
282.        print('X scored %s points. O scored %s points.' % (scores['X'],
scores['O']))
283.
284.        if scores['X'] > scores['O']:
285.            xwins += 1
286.        elif scores['X'] < scores['O']:
287.            owins += 1
288.        else:
289.            ties += 1
290.
291. numGames = float(numGames)
292. xpercent = round(((xwins / numGames) * 100), 2)
293. opercent = round(((owins / numGames) * 100), 2)
294. tiepercent = round(((ties / numGames) * 100), 2)
295. print('X wins %s games (%s%%), O wins %s games (%s%%), ties for %s games
(%s%%) of %s games total.' % (xwins, xpercent, owins, opercent, ties,
tiepercent, numGames))
```

How the AISim2.py Code Works

You have added the variables xwins, owins, and ties to lines 248 to 250 to keep track of how many times X wins, O wins, and when they tie. Lines 284 to 289 increment these variables at the end of each game, before it loops back to start a new game.

You have removed most of the print() function calls from the program, as well as the calls to drawBoard(). When you run *AISim2.py*, it asks you how many games you want to run. Now that you've taken out the call to drawBoard() and replace the while True: loop with a for game in range(numGames): loop, you can run a number of games without stopping for the user to type anything. Here is a sample run of ten of computer vs. computer Reversi games:

```
Welcome to Reversi!
Enter number of games to run: 10
Game #0: X scored 40 points. O scored 23 points.
Game #1: X scored 24 points. O scored 39 points.
Game #2: X scored 31 points. O scored 30 points.
Game #3: X scored 41 points. O scored 23 points.
Game #4: X scored 30 points. O scored 34 points.
Game #5: X scored 37 points. O scored 27 points.
Game #6: X scored 29 points. O scored 33 points.
Game #7: X scored 31 points. O scored 33 points.
Game #8: X scored 32 points. O scored 32 points.
Game #9: X scored 41 points. O scored 22 points.
X wins 5 games (50.0%), O wins 4 games (40.0%), ties for 1 games (10.0%) of
10.0 games total.
```

Because the algorithms include randomness, your run won't have the exact numbers as above.

Printing things out to the screen slows the computer down, but now that you have removed that code, the computer can run an entire game of Reversi in about a second or two. Think about it. Each time the program printed out one of those lines with the final score, it ran through an entire game (which is about fifty or sixty moves, each move carefully checked to be the one that gets the most points).
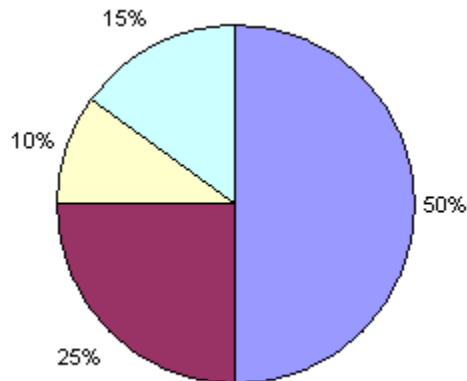
Percentages

Figure 16-1: A pie chart with 10%, 15%, 25%, and 50% portions.

Percentages are a portion of a total amount, and range from 0% to 100%. If you had 100% of a pie, you would have the entire pie. If you had 0% of a pie, you wouldn't have any pie at all. 50% of the pie would be half of the pie. A pie is a common image to use for percentages. In fact,

there's a kind of chart called a **pie chart** which shows how much of the full total a certain portion is. Figure 16-1 is a pie chart with 10%, 15%, 25%, and 50% portions below. Notice that 10% + 15% + 25% + 50% adds up to 100%: a whole pie.

We can calculate the percentage with division. To get a percentage, divide the part you have by the total, and then multiply by one hundred. For example, if X won 50 out of 100 games, you would calculate the expression 50 / 100, which would evaluate to 0.5. Multiply this by 100 to get a percentage (in this case, 50%).

Notice that if X won 100 out of 200 games, you could calculate the percentage with 100 / 200, which would also evaluate to 0.5. When you multiply 0.5 by 100 to get the percentage, you get 50%. Winning 100 out of 200 games is the same percentage (that is, the same portion) as winning 50 out of 100 games.

Division Evaluates to Floating Point

It is important to note that when you use the / division operator, the expression will always evaluate to a floating point number. For example, the expression 10 / 2 will evaluate to the floating point value 5.0, not to the integer value 5.

This is important to remember, because adding an integer to a floating point value with the + addition operator will also always evaluate to a floating point value. For example, 3 + 4.0 will evaluate to the floating point value 7.0 and not to the integer 7.

Try entering the following code into the interactive shell:

```
>>> spam = 100 / 4
>>> spam
25.0
>>> spam = spam + 20
>>> spam
45.0
```

Notice that in the above example, the data type of the value stored in spam is always a floating point value. You can pass the floating point value to the int() function, which will return an integer form of the floating point value. But this will always round the floating point value down. For example, the expressions int(4.0), int(4.2), and int(4.9) will all evaluate to 4, and never 5.

The round() function

The round() function will round a float number to the nearest whole float number. Try entering the following into the interactive shell:

```
>>> round(10.0)
10.0
>>> round(10.2)
10.0
>>> round(8.7)
9.0
>>> round(3.4999)
3.0
>>> round(2.5422, 2)
2.54
```

The `round()` function also has an optional parameter, where you can specify to what place you want to round the number to. For example, the expression `round(2.5422, 2)` evaluates to `2.54` and `round(2.5422, 3)` evaluates to `2.542`.

Displaying the Statistics

```
291. numGames = float(numGames)
292. xpercent = round(((xwins / numGames) * 100), 2)
293. opercent = round(((owins / numGames) * 100), 2)
294. tiepercent = round(((ties / numGames) * 100), 2)
295. print('X wins %s games (%s%%), O wins %s games (%s%%), ties for %s games
(%s%%) of %s games total.' % (xwins, xpercent, owins, opercent, ties,
tiepercent, numGames))
```

The code at the bottom of the program will show the user how many wins X and O had, how many ties there were, and how what percentages these make up. Statistically, the more games you run, the more accurate your percentages will be for finding the best AI algorithm. If you only ran ten games, and X won three of them, then it would seem that X's algorithm only wins 30% of the time. However, if you run a hundred, or even a thousand games, then you may find that X's algorithm wins closer to 50% (that is, half) of the games.

To find the percentages, divide the number of wins or ties by the total number of games. Then multiply the result by 100. However, you may end up with a number like `66.66666666666667`. So pass this number to the `round()` function with the second parameter of 2 to limit the precision to two decimal places, so it will return a float like `66.67` instead (which is much more readable).

Let's try another experiment. Run *AISim2.py* again, but this time have it run a hundred games:

Sample Run of AISim2.py

```
Welcome to Reversi!
Enter number of games to run: 100
```

```
Game #0: X scored 42 points. O scored 18 points.
Game #1: X scored 26 points. O scored 37 points.
Game #2: X scored 34 points. O scored 29 points.
Game #3: X scored 40 points. O scored 24 points.
...skipped for brevity...
Game #96: X scored 22 points. O scored 39 points.
Game #97: X scored 38 points. O scored 26 points.
Game #98: X scored 35 points. O scored 28 points.
Game #99: X scored 24 points. O scored 40 points.
X wins 46 games (46.0%), O wins 52 games (52.0%), ties for 2 games (2.0%) of
100.0 games total.
```

Depending on how fast your computer is, this run might have taken a about a couple minutes. You can see that the results of all one hundred games still evens out to about fifty-fifty, because both X and O are using the same algorithm to win.

Comparing Different AI Algorithms

Let's add some new functions with new algorithms. But first click on **File ► Save As**, and save this file as *AISim3.py*. Before the print('Welcome to Reversi!') line, add these functions in the following source code listing.

AISim3.py

If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/AISim3.

AISim3.py
```
245. def getRandomMove(board, tile):
246.     # Return a random move.
247.     return random.choice( getValidMoves(board, tile) )
248.
249.
250. def isOnSide(x, y):
251.     return x == 0 or x == 7 or y == 0 or y ==7
252.
253.
254. def getCornerSideBestMove(board, tile):
255.     # Return a corner move, or a side move, or the best move.
256.     possibleMoves = getValidMoves(board, tile)
257.
258.     # randomize the order of the possible moves
259.     random.shuffle(possibleMoves)
260.
```

```
261.        # always go for a corner if available.
262.        for x, y in possibleMoves:
263.            if isOnCorner(x, y):
264.                return [x, y]
265.
266.        # if there is no corner, return a side move.
267.        for x, y in possibleMoves:
268.            if isOnSide(x, y):
269.                return [x, y]
270.
271.        return getComputerMove(board, tile)
272.
273.
274. def getSideBestMove(board, tile):
275.        # Return a corner move, or a side move, or the best move.
276.        possibleMoves = getValidMoves(board, tile)
277.
278.        # randomize the order of the possible moves
279.        random.shuffle(possibleMoves)
280.
281.        # return a side move, if available
282.        for x, y in possibleMoves:
283.            if isOnSide(x, y):
284.                return [x, y]
285.
286.        return getComputerMove(board, tile)
287.
288.
289. def getWorstMove(board, tile):
290.        # Return the move that flips the least number of tiles.
291.        possibleMoves = getValidMoves(board, tile)
292.
293.        # randomize the order of the possible moves
294.        random.shuffle(possibleMoves)
295.
296.        # Go through all the possible moves and remember the best scoring move
297.        worstScore = 64
298.        for x, y in possibleMoves:
299.            dupeBoard = getBoardCopy(board)
300.            makeMove(dupeBoard, tile, x, y)
301.            score = getScoreOfBoard(dupeBoard)[tile]
302.            if score < worstScore:
303.                worstMove = [x, y]
304.                worstScore = score
305.
306.        return worstMove
307.
```

```
308.
309. def getCornerWorstMove(board, tile):
310.     # Return a corner, a space, or the move that flips the least number of
tiles.
311.     possibleMoves = getValidMoves(board, tile)
312.
313.     # randomize the order of the possible moves
314.     random.shuffle(possibleMoves)
315.
316.     # always go for a corner if available.
317.     for x, y in possibleMoves:
318.         if isOnCorner(x, y):
319.             return [x, y]
320.
321.     return getWorstMove(board, tile)
322.
323.
324.
325. print('Welcome to Reversi!')
```

How the AISim3.py Code Works

A lot of these functions are similar to one another, and some of them use the new isOnSide()
function. Here's a review of the new algorithms we've made:

Table 17-1: Functions used for our Reversi AI.

Function	Description
getRandomMove()	Randomly choose a valid move to make.
getCornerSideBestMove()	Take a corner move if available. If there's no corner, take a space on the side. If no sides are available, use the regular getComputerMove() algorithm.
getSideBestMove()	Take a side space if there's one available. If not, then use the regular getComputerMove() algorithm. This means side spaces are chosen before corner spaces.
getWorstMove()	Take the space that will result in the fewest tiles being flipped.
getCornerWorstMove()	Take a corner space, if available. If not, use the getWorstMove() algorithm.

Comparing the Random Algorithm Against the Regular Algorithm

Now the only thing to do is replace one of the getComputerMove() calls in the main part of the program with one of the new functions. Then you can run several games and see how often one algorithm wins over the other. First, let's replace O's algorithm with the one in getComputerMove() with getRandomMove() on line 351:

```
351.               x, y = getRandomMove(mainBoard, 'O')
```

When you run the program with a hundred games now, it will look something like this:

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 25 points. O scored 38 points.
Game #1: X scored 32 points. O scored 32 points.
Game #2: X scored 15 points. O scored 0 points.

...skipped for brevity...

Game #97: X scored 41 points. O scored 23 points.
Game #98: X scored 33 points. O scored 31 points.
Game #99: X scored 45 points. O scored 19 points.
X wins 84 games (84.0%), O wins 15 games (15.0%), ties for 1 games (1.0%) of
100.0 games total.
```

Wow! X won far more often than O did. That means that the algorithm in getComputerMove() (take any available corners, otherwise take the space that flips the most tiles) wins more games than the algorithm in getRandomMove() (which makes moves randomly). This makes sense, because making intelligent choices is usually better than just choosing things at random.

Comparing the Random Algorithm Against Itself

What if we changed O's algorithm to also use the algorithm in getRandomMove()? Let's find out by changing O's function call on line 351 from getComputerMove() to getRandomMove() and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 37 points. O scored 24 points.
Game #1: X scored 19 points. O scored 45 points.

...skipped for brevity...

Game #98: X scored 27 points. O scored 37 points.
```

```
Game #99: X scored 38 points. O scored 22 points.
X wins 42 games (42.0%), O wins 54 games (54.0%), ties for 4 games (4.0%) of
100.0 games total.
```

As you can see, when both players are making random moves, they each win about 50% of the time. (In the above case, O happen to get lucky and won a little bit more than half of the time.)

Just like moving on the corner spaces is a good idea because they cannot be flipped, moving on the side spaces may also be a good idea. On the side, the tile has the edge of the board and isn't as out in the open as the other pieces. The corners are still preferable to the side spaces, but moving on the sides (even when there's a move that can flip more pieces) may be a good strategy.

Comparing the Regular Algorithm Against the CornersSideBest Algorithm

Change X's algorithm on line 346 to use `getComputerMove()` (the original algorithm) and O's algorithm on line 351 to use `getCornerSideBestMove()` (which first tries to move on a corner, then tries to move on a side space, and then takes the best remaining move), and let's run a hundred games to see which is better. Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 52 points. O scored 12 points.
Game #1: X scored 10 points. O scored 54 points.

...skipped for brevity...

Game #98: X scored 41 points. O scored 23 points.
Game #99: X scored 46 points. O scored 13 points.
X wins 65 games (65.0%), O wins 31 games (31.0%), ties for 4 games (4.0%) of
100.0 games total.
```

Wow! That's unexpected. It seems that choosing the side spaces over a space that flips more tiles is a bad strategy to use. The benefit of the side space isn't greater than the cost of flipping fewer of the opponent's tiles. Can we be sure of these results? Let's run the program again, but this time play one thousand games. This may take a few minutes for your computer to run (but it would take weeks for you to do this by hand!) Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 1000
Game #0: X scored 20 points. O scored 44 points.
Game #1: X scored 54 points. O scored 9 points.
```

```
...skipped for brevity...

Game #998: X scored 38 points. O scored 23 points.
Game #999: X scored 38 points. O scored 26 points.
X wins 611 games (61.1%), O wins 363 games (36.3%), ties for 26 games (2.6%) of
1000.0 games total.
```

The more accurate statistics from the thousand-games run are about the same as the statistics from the hundred-games run. It seems that choosing the move that flips the most tiles is a better idea than choosing a side move.

Comparing the Regular Algorithm Against the Worst Algorithm

Now set the X player's algorithm on line 346 to use `getComputerMove()` and the O player's algorithm on line 351 to `getWorstMove()` (which makes the move that flips over the least number of tiles), and run a hundred games. Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 50 points. O scored 14 points.
Game #1: X scored 38 points. O scored 8 points.
...skipped for brevity...
Game #98: X scored 36 points. O scored 16 points.
Game #99: X scored 19 points. O scored 0 points.
X wins 98 games (98.0%), O wins 2 games (2.0%), ties for 0 games (0.0%) of
100.0 games total.
```

Whoa! The algorithm in `getWorstMove()`, which always chose the move that flips the fewest tiles, will almost always lose to the regular algorithm. This isn't really surprising at all. (In fact, it's surprising that this strategy wins even 2% of the time!)

Comparing the Regular Algorithm Against the WorstCorner Algorithm

How about when we replace `getWorstMove()` on line 351 with `getCornerWorstMove()`? This is the same algorithm except it takes any available corner pieces before taking the worst move. Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 36 points. O scored 7 points.
Game #1: X scored 44 points. O scored 19 points.

...skipped for brevity...
```

```
Game #98: X scored 47 points. O scored 17 points.
Game #99: X scored 36 points. O scored 18 points.
X wins 94 games (94.0%), O wins 6 games (6.0%), ties for 0 games (0.0%) of
100.0 games total.
```

The `getCornerWorstMove()` still loses most of the games, but it seems to win a few more games than `getWorstMove()` (6% compared to 2%). Does taking the corner spaces when they are available really make a difference?

Comparing the Worst Algorithm Against the WorstCorner Algorithm

You can check by setting X's algorithm to `getWorstMove()` and O's algorithm to `getCornerWorstMove()`, and then running the program. Try changing the function calls and running the program again.

```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 25 points. O scored 39 points.
Game #1: X scored 26 points. O scored 33 points.

...skipped for brevity...

Game #98: X scored 36 points. O scored 25 points.
Game #99: X scored 29 points. O scored 35 points.
X wins 32 games (32.0%), O wins 67 games (67.0%), ties for 1 games (1.0%) of
100.0 games total.
```

Yes, even when otherwise making the worst move, it does seem like taking the corners results in many more wins. While you've found out that going for the sides makes you lose more often, going for the corners is always a good idea.

Summary

This chapter didn't really cover a game, but it modeled various strategies for Reversi. If we thought that taking side moves in Reversi was a good idea, we would have to spend weeks, even months, carefully playing games of Reversi by hand and writing down the results. But if we know how to program a computer to play Reversi, then we can have the computer play Reversi using these strategies for us. If you think about it, you'll realize that the computer is executing millions of lines of our Python program in seconds! Your experiments with the simulation of Reversi can help you learn more about playing Reversi in real life.

In fact, this chapter would make a good science fair project. Your problem can be which set of moves leads to the most wins against other sets of moves, and make a hypothesis about which is

the best strategy. After running several simulations, you can determine which strategy works best. With programming you can make a science fair project out of a simulation of any board game! And it is all because you know how to instruct the computer to do it, step by step, line by line. You can speak the computer's language, and get it to do large amounts of data processing and number crunching for you.

That's all for the text-based games in this book. Games that only use text can be fun, even though they're simple. But most modern games use graphics, sound, and animation to make much more exciting looking games. For the rest of the chapters in this book, we will learn how to create games with graphics by using a Python module called Pygame.

Chapter 17

GRAPHICS AND ANIMATION

Topics Covered In This Chapter:
- Installing Pygame
- Colors and Fonts in Pygame
- Aliased and Anti-Aliased Graphics
- Attributes
- The `pygame.font.Font`, `pygame.Surface`, `pygame.Rect`, and `pygame.PixelArray` Data Types
- Constructor Functions
- Pygame's Drawing Functions
- The `blit()` Method for Surface Objects
- Events
- Animation

So far, all of our games have only used text. Text is displayed on the screen as output, and the player types in text from the keyboard as input. Just using text makes programming easy to learn. But in this chapter, we'll make some more exciting games with advanced graphics and sound using the Pygame module.

Chapters 17, 18, and 19 teaches you how to use Pygame to make games with graphics, animation, mouse input, and sound. In these chapters we'll write source code for simple programs that are not games but demonstrate the Pygame concepts we've learned. The game in Chapter 20 will use all these concepts together to create a game.

Installing Pygame

Pygame doesn't come with Python. Like Python, Pygame is free to download. In a web browser, go to the URL http://invpy.org/downloadpygame and download the Pygame installer file for your operating system and version of Python.

Open the installer file after downloading it, and follow the instructions until Pygame has finished installing. To check that Pygame installed correctly, type the following into the interactive shell:

```
>>> import pygame
```

If nothing appears after you hit the ENTER key, then you know Pygame was successfully installed. If the error ImportError: No module named pygame appears, try to install Pygame again (and make sure you typed import pygame correctly).

Figure 17-1: The pygame.org website.

The Pygame website at http://pygame.org has information on how to use Pygame, as well as several other game programs made with Pygame. Figure 17-1 shows the Pygame website.

Hello World in Pygame

The first Pygame program is a new "Hello World!" program like you created at the beginning of the book. This time, you'll use Pygame to make "Hello world!" appear in a graphical window instead of as text.

Pygame doesn't work well with the interactive shell. Because of this, you can only write Pygame programs and cannot send instructions to Pygame one at a time through the interactive shell.

Pygame programs also do not use the input() function. There is no text input and output. Instead, the program displays output in a window by drawing graphics and text to the window. Pygame program's input comes from the keyboard and the mouse through things called events. Events are explained in the next chapter.

Source Code of Hello World

Type in the following code into the file editor, and save it as *pygameHelloWorld.py*. If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/pygameHelloWorld.

```
                                                    pygameHelloWorld.py
 1. import pygame, sys
 2. from pygame.locals import *
 3.
 4. # set up pygame
 5. pygame.init()
 6.
 7. # set up the window
 8. windowSurface = pygame.display.set_mode((500, 400), 0, 32)
 9. pygame.display.set_caption('Hello world!')
10.
11. # set up the colors
12. BLACK = (0, 0, 0)
13. WHITE = (255, 255, 255)
14. RED = (255, 0, 0)
15. GREEN = (0, 255, 0)
16. BLUE = (0, 0, 255)
17.
18. # set up fonts
19. basicFont = pygame.font.SysFont(None, 48)
20.
21. # set up the text
22. text = basicFont.render('Hello world!', True, WHITE, BLUE)
23. textRect = text.get_rect()
24. textRect.centerx = windowSurface.get_rect().centerx
25. textRect.centery = windowSurface.get_rect().centery
26.
27. # draw the white background onto the surface
28. windowSurface.fill(WHITE)
29.
30. # draw a green polygon onto the surface
31. pygame.draw.polygon(windowSurface, GREEN, ((146, 0), (291, 106), (236,
277), (56, 277), (0, 106)))
32.
33. # draw some blue lines onto the surface
34. pygame.draw.line(windowSurface, BLUE, (60, 60), (120, 60), 4)
35. pygame.draw.line(windowSurface, BLUE, (120, 60), (60, 120))
36. pygame.draw.line(windowSurface, BLUE, (60, 120), (120, 120), 4)
37.
38. # draw a blue circle onto the surface
39. pygame.draw.circle(windowSurface, BLUE, (300, 50), 20, 0)
40.
41. # draw a red ellipse onto the surface
42. pygame.draw.ellipse(windowSurface, RED, (300, 250, 40, 80), 1)
43.
44. # draw the text's background rectangle onto the surface
```

```
45. pygame.draw.rect(windowSurface, RED, (textRect.left - 20, textRect.top -
20, textRect.width + 40, textRect.height + 40))
46.
47. # get a pixel array of the surface
48. pixArray = pygame.PixelArray(windowSurface)
49. pixArray[480][380] = BLACK
50. del pixArray
51.
52. # draw the text onto the surface
53. windowSurface.blit(text, textRect)
54.
55. # draw the window onto the screen
56. pygame.display.update()
57.
58. # run the game loop
59. while True:
60.     for event in pygame.event.get():
61.         if event.type == QUIT:
62.             pygame.quit()
63.             sys.exit()
```

Running the Hello World Program

When you run this program, you should see a new window appear which looks like Figure 17-2.

What is nice about using a window instead of a console is that the text can appear anywhere in the window, not just after the previous text you have printed. The text can be any color and size. The window is like a blank painting canvas, and you can draw whatever you like on it.

Importing the Pygame Module

Let's go over each of these lines of code and find out what they do.

```
1. import pygame, sys
2. from pygame.locals import *
```
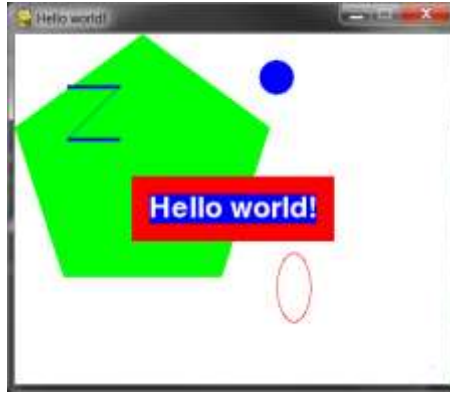
Figure 17-2: The "Hello World" program.

First you need to import the pygame module so you can call Pygame's functions. You can import several modules on the same line by delimiting the module names with commas. Line 1 imports both the pygame and sys modules.

The second line imports the pygame.locals module. This module contains many constant variables that you'll use with Pygame such as QUIT or K_ESCAPE (explained later). However, using the form from moduleName import * you can import the pygame.locals module but not have to type pygame.locals in front of the module's constants.

If you have from sys import * instead of import sys in your program, you could call exit() instead of sys.exit() in your code. But most of the time it is better to use the full function name so you know which module the function is in.

The pygame.init() Function

```
4. # set up pygame
5. pygame.init()
```

All Pygame programs must call the pygame.init() after importing the pygame module but before calling any other Pygame functions. This perform's Pygame's necessary initialization steps.

Tuples

Tuple values are similar to lists, except they use parentheses instead of square brackets. Also, like strings, tuples cannot be modified. For example, try entering the following into the interactive shell:

```
>>> spam = ('Life', 'Universe', 'Everything', 42)
```

```
>>> spam[0]
'Life'
>>> spam[3]
42
>>> spam[1:3]
('Universe', 'Everything')
```

The `pygame.display.set_mode()` and `pygame.display.set_caption()` Functions

```
7. # set up the window
8. windowSurface = pygame.display.set_mode((500, 400), 0, 32)
9. pygame.display.set_caption('Hello world!')
```

Line 8 creates a GUI window by calling the `set_mode()` method in the `pygame.display` module. (The `display` module is a module inside the `pygame` module. Even the `pygame` module has its own modules!)

A pixel is the tiniest dot on your computer screen. A single pixel on your screen can light up into any color. All the pixels on your screen work together to display all the pictures you see. To create a window 500 pixels wide and 400 pixels high, use the tuple `(500, 400)` for the first parameter to `pygame.display.set_mode()`.

There are three parameters to the `set_mode()` method. The first is a tuple of two integers for the width and height of the window, in pixels. The second and third options are advanced options that are beyond the scope of this book. Just pass `0` and `32` for them, respectively.

The `set_mode()` function returns a `pygame.Surface` object (which we will call `Surface` objects for short). **Objects** is just another name for a value of a data type that has methods. For example, strings are objects in Python because they have data (the string itself) and methods (such as `lower()` and `split()`). The `Surface` object represents the window.

Variables store references to objects just like they store reference for lists and dictionaries. The References section in Chapter 10 explains references.

RGB Colors

```
11. # set up the colors
12. BLACK = (0, 0, 0)
13. WHITE = (255, 255, 255)
14. RED = (255, 0, 0)
15. GREEN = (0, 255, 0)
16. BLUE = (0, 0, 255)
```

Table 17-1: Colors and their RGB values.

Color	RGB Values
Black	(0, 0, 0)
Blue	(0, 0, 255)
Gray	(128, 128, 128)
Green	(0, 128, 0)
Lime	(0, 255, 0)
Purple	(128, 0, 128)
Red	(255, 0, 0)
Teal	(0, 128, 128)
White	(255, 255, 255)
Yellow	(255, 255, 0)

There are three primary colors of light: red, green and blue. By combining different amounts of these three colors (which is what your computer screen does), you can form any other color. In Pygame, tuples of three integers are the data structures that represent a color. These are called **RGB Color** values.

The first value in the tuple is how much red is in the color. A value of 0 means there's no red in this color, and a value of 255 means there's a maximum amount of red in the color. The second value is for green and the third value is for blue. These three integers form an RGB tuple.

For example, the tuple (0, 0, 0) has no amount of red, green, or blue. The resulting color is completely black. The tuple (255, 255, 255) has a maximum amount of red, green, and blue, resulting in white.

The tuple (255, 0, 0) represents the maximum amount of red but no amount of green and blue, so the resulting color is red. Similarly, (0, 255, 0) is green and (0, 0, 255) is blue.

You can mix the amount of red, green, and blue to get any shade of any color. Table 17-1 has some common colors and their RGB values. The web page http://invpy.com/colors also lists several more tuple values for different colors.

Fonts, and the `pygame.font.SysFont()` Function

```
18. # set up fonts
19. basicFont = pygame.font.SysFont(None, 48)
```

Programming is Fun!

Programming is Fun!

PROGRAMMING IS FUN!

Programming is Fun!

Programming is Fun!

Programming is Fun!

Figure 17-3: Examples of different fonts.

A font is a complete set of letters, numbers, symbols, and characters drawn in a single style. Figure 17-3 shows the same sentence printed in different fonts.

In our earlier games, we only told Python to print text. The color, size, and font that was used to display this text was completely determined by your operating system. The Python program couldn't change the font. However, Pygame can draw text in any font on your computer.

Line 19 creates a `pygame.font.Font` object (called Font objects for short) by calling the `pygame.font.SysFont()` function. The first parameter is the name of the font, but we will pass the `None` value to use the default system font. The second parameter is the size of the font (which is measured in units called *points*).

The `render()` Method for `Font` Objects

```
21. # set up the text
22. text = basicFont.render('Hello world!', True, WHITE, BLUE)
23. textRect = text.get_rect()
```
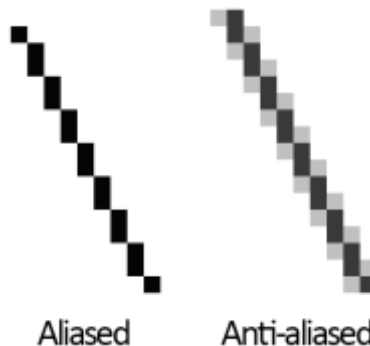
Aliased   Anti-aliased

Figure 17-4: An enlarged view of an aliased line and an anti-aliased line.

The `Font` object that you've stored in the `basicFont` variable has a method called `render()`. This method will return a `Surface` object with the text drawn on it. The first parameter to `render()` is the string of the text to draw. The second parameter is a Boolean for whether or not you want anti-aliasing.

On line 22, pass `True` to use anti-aliasing. Anti-aliasing blurs your text slightly to make it look smoother. Figure 17-4 shows what a line (with enlarged pixels) looks like with and without anti-aliasing.

Attributes

```
24. textRect.centerx = windowSurface.get_rect().centerx
25. textRect.centery = windowSurface.get_rect().centery
```

The `pygame.Rect` data type (called Rect for short) represent rectangular areas of a certain size and location. To create a new Rect object call the function `pygame.Rect()`. The parameters are integers for the XY coordinates of the top left corner, followed by the width and height, all in pixels.

The function name with the parameters looks like this: `pygame.Rect(left, top, width, height)`

Just like methods are functions that are associated with an object, **attributes** are variables that are associated with an object. The Rect data type has many attributes that describe the rectangle they represent. Table 17-2 is a list of attributes of a Rect object named `myRect`.

The great thing about Rect objects is that if you modify any of these attributes, all the other attributes will automatically modify themselves also. For example, if you create a Rect object that is 20 pixels wide and 20 pixels high, and has the top left corner at the coordinates (30, 40), then the X-coordinate of the right side will automatically be set to `50` (because 20 + 30 = 50).

However, if you change the `left` attribute with the line `myRect.left = 100`, then Pygame will automatically change the `right` attribute to 120 (because 20 + 100 = 120). Every other attribute for that Rect object is also updated.

The `get_rect()` Methods for `pygame.font.Font` and `pygame.Surface` Objects

Notice that both the Font object (stored in the `text` variable on line 23) and the Surface object (stored in `windowSurface` variable on line 24) both have a method called `get_rect()`. Technically, these are two different methods. But the programmers of Pygame gave them the same name because they both do the same thing and return Rect objects that represent the size and position of the Font or Surface object.

The module you import is `pygame`, and inside the `pygame` module are the `font` and `surface` modules. Inside those modules are the `Font` and `Surface` data types. The Pygame programmers made the modules begin with a lowercase letter, and the data types begin with an uppercase letter. This makes it easier to distinguish the data types and the modules.

Constructor Functions

Create a `pygame.Rect` object by calling a function named `pygame.Rect()`. The `pygame.Rect()` function has the same name as the `pygame.Rect` data type. Functions that have the same name as their data type and create objects or values of this data type are called **constructor functions**.

The *fill()* Method for Surface Objects

```
27. # draw the white background onto the surface
28. windowSurface.fill(WHITE)
```

You want to fill the entire surface stored in `windowSurface` with the color white. The `fill()` function will completely cover the entire surface with the color you pass as the parameter. (In this case, the `WHITE` variable is set to the value (255, 255, 255).)

An important thing to know about Pygame is that the window on the screen won't change when you call the `fill()` method or any of the other drawing functions. These will change the Surface object, but the Surface object won't be drawn on the screen until the `pygame.display.update()` function is called.

This is because modifying the Surface object in the computer's memory is much faster than modifying the image on the screen. It is much more efficient to draw onto the screen once after all of the drawing functions have drawn to the surface.

Pygame's Drawing Functions

The *pygame.draw.polygon()* Function

```
30. # draw a green polygon onto the surface
31. pygame.draw.polygon(windowSurface, GREEN, ((146, 0), (291, 106), (236,
277), (56, 277), (0, 106)))
```

A polygon is multisided shape with straight line sides. Circles and ellipses are not polygons. Figure 17-5 has some examples of polygons.

Table 17-2: Rect Attributes

`pygame.Rect` Attribute	Description
`myRect.left`	Integer value of the X-coordinate of the left side of the rectangle.
`myRect.right`	Integer value of the X-coordinate of the right side of the rectangle.
`myRect.top`	Integer value of the Y-coordinate of the top side of the rectangle.
`myRect.bottom`	Integer value of the Y-coordinate of the bottom side of the rectangle.
`myRect.centerx`	Integer value of the X-coordinate of the center of the rectangle.
`myRect.centery`	Integer value of the Y-coordinate of the center of the rectangle.
`myRect.width`	Integer value of the width of the rectangle.
`myRect.height`	Integer value of the height of the rectangle.
`myRect.size`	A tuple of two integers: (width, height)
`myRect.topleft`	A tuple of two integers: (left, top)
`myRect.topright`	A tuple of two integers: (right, top)
`myRect.bottomleft`	A tuple of two integers: (left, bottom)
`myRect.bottomright`	A tuple of two integers: (right, bottom)
`myRect.midleft`	A tuple of two integers: (left, centery)
`myRect.midright`	A tuple of two integers: (right, centery)
`myRect.midtop`	A tuple of two integers: (centerx, top)
`myRect.midbottom`	A tuple of two integers: (centerx, bottom)

Figure 17-5: Examples of Polygons.

The `pygame.draw.polygon()` function can draw any polygon shape you give it. The parameters, in order, are:

- The Surface object to draw the polygon on.

- The color of the polygon.

- A tuple of tuples that represents the XY coordinates of the points to draw in order. The last tuple will automatically connect to the first tuple to complete the shape.

- Optionally, an integer for the width of the polygon lines. Without this, the polygon will be filled in.

Line 31 draws a green pentagon on the Surface object.

The `pygame.draw.line()` Function

```
33. # draw some blue lines onto the surface
34. pygame.draw.line(windowSurface, BLUE, (60, 60), (120, 60), 4)
35. pygame.draw.line(windowSurface, BLUE, (120, 60), (60, 120))
36. pygame.draw.line(windowSurface, BLUE, (60, 120), (120, 120), 4)
```

The parameters, in order, are:

- The Surface object to draw the line on.

- The color of the line.

- A tuple of two integers for the XY coordinate of one end of the line.

- A tuple of two integers for the XY coordinates of the other end of the line.

- Optionally, an integer for the width of the line.

If you pass 4 for the width, the line will be four pixels thick. If you do not specify the `width` parameter, it will take on the default value of 1. The three `pygame.draw.line()` calls on lines 34, 35, and 36 draw the blue "Z" on the Surface object.

The `pygame.draw.circle()` Function

```
38. # draw a blue circle onto the surface
39. pygame.draw.circle(windowSurface, BLUE, (300, 50), 20, 0)
```

The parameters, in order, are:

- The Surface object to draw the circle on.

- The color of the circle.

- A tuple of two integers for the XY coordinate of the center of the circle.

- An integer for the radius (that is, the size) of the circle.

- Optionally, an integer for the width. A width of 0 means that the circle will be filled in.

Line 39 draws a blue circle on the Surface object.

The `pygame.draw.ellipse()` Function

```
41. # draw a red ellipse onto the surface
42. pygame.draw.ellipse(windowSurface, RED, (300, 250, 40, 80), 1)
```

The `pygame.draw.ellipse()` function is similar to the `pygame.draw.circle()` function. The parameters, in order, are:

- The Surface object to draw the ellipse on.

- The color of the ellipse.

- A tuple of four integers is passed for the left, top, width, and height of the ellipse.

- Optionally, an integer for the width. A width of 0 means that the circle will be filled in.

Line 42 draws a red ellipse on the Surface object.

The `pygame.draw.rect()` Function

```
44. # draw the text's background rectangle onto the surface
45. pygame.draw.rect(windowSurface, RED, (textRect.left - 20, textRect.top -
20, textRect.width + 40, textRect.height + 40))
```

The `pygame.draw.rect()` function will draw a rectangle. The third parameter is a tuple of four integers for the left, top, width, and height of the rectangle. Instead of a tuple of four integers for the third parameter, you can also pass a Rect object.

On line 45, you want the rectangle you draw to be 20 pixels around all the sides of the text. This is why you want the drawn rectangle's left and top to be the left and top of `textRect` minus 20. (Remember, you subtract because coordinates decrease as you go left and up.) And the width and

height are equal to the width and height of the `textRect` plus 40 (because the left and top were moved back 20 pixels, so you need to make up for that space).

The *pygame.PixelArray* Data Type

```
47. # get a pixel array of the surface
48. pixArray = pygame.PixelArray(windowSurface)
49. pixArray[480][380] = BLACK
```

Line 48 creates a `pygame.PixelArray` object (called a PixelArray object for short). The PixelArray object is a list of lists of color tuples that represents the Surface object you passed it.

Line 48 passes `windowSurface` to the `pygame.PixelArray()` call, so assigning `BLACK` to `pixArray[480][380]` on line 49 will change the pixel at the coordinates (480, 380) to be a black pixel. Pygame will automatically modify the `windowSurface` object with this change.

The first index in the PixelArray object is for the X-coordinate. The second index is for the Y-coordinate. PixelArray objects make it easy to set individual pixels on a PixelArray object to a specific color.

```
50. del pixArray
```

Creating a PixelArray object from a Surface object will lock that Surface object. Locked means that no `blit()` function calls (described next) can be made on that Surface object. To unlock the Surface object, you must delete the PixelArray object with the `del` operator. If you forget to delete the PixelArray object, you'll get an error message that says `pygame.error: Surfaces must not be locked during blit`.

The *blit()* Method for Surface Objects

```
52. # draw the text onto the surface
53. windowSurface.blit(text, textRect)
```

The `blit()` method will draw the contents of one Surface object onto another Surface object. Line 54 will draw the "Hello world!" Surface object in `text` and draws it to the Surface object stored in the `windowSurface` variable.

The second parameter to `blit()` specifies where on the windowSurface surface the text surface should be drawn. Pass the Rect object you got from calling `text.get_rect()` on line 23.

The `pygame.display.update()` Function

```
55. # draw the window onto the screen
56. pygame.display.update()
```

In Pygame, nothing is actually drawn to the screen until the `pygame.display.update()` function is called. This is because drawing to the screen is slow compared to drawing on the Surface objects in the computer's memory. You do not want to update to the screen after each drawing function is called, but only update the screen once after all the drawing functions have been called.

Events and the Game Loop

In previous games, all of the programs print everything immediately until they reach a `input()` function call. At that point, the program stops and waits for the user to type something in and press ENTER. But Pygame programs are constantly running through a loop called the **game loop**. In this program, all the lines of code in the game loop execute about a hundred times a second.

The game loop is a loop that constantly checks for new events, updates the state of the window, and draws the window on the screen. **Events** are objects of the `pygame.event.Event` data type that are generated by Pygame whenever the user presses a key, clicks or moves the mouse, or makes some other event occur. (These events are listed on Table 18-1.)

```
58. # run the game loop
59. while True:
```

Line 59 is the start of the game loop. The condition for the `while` statement is set to `True` so that it loops forever. The only time the loop exits is if an event causes the program to terminate.

The `pygame.event.get()` Function

```
60.     for event in pygame.event.get():
61.         if event.type == QUIT:
```

Calling `pygame.event.get()` retrieves any new `pygame.event.Event` objects (called Event objects for short) that have been generated since the last call to `pygame.event.get()`. These events are returned as a list of Event objects. All Event objects have an attribute called `type` which tell us what type of event it is. (In this chapter we only deal with the `QUIT` types of event. The other types of events are covered in the next chapter.)

Line 60 has a `for` loop to iterate over each Event object in the list returned by `pygame.event.get()`. If the `type` attribute of the event is equal to the constant variable `QUIT`, then you know the user has closed the window and wants to terminate the program.

Pygame generates the `QUIT` event (which was imported from the `pygame.locals` module) when the user clicks on the close button (usually an ×) of the program's window. It is also generated if the computer is shutting down and tries to terminate all the running programs. For whatever reason the `QUIT` event was generated, you should terminate the program.

The `pygame.quit()` Function

```
62.            pygame.quit()
63.            sys.exit()
```

If the `QUIT` event has been generated, the program should call both `pygame.quit()` and `sys.exit()`.

This has been the simple "Hello world!" program from Pygame. We've covered many new topics that we didn't have to deal with in our previous games. Even though the code is more complicated, the Pygame programs can also be much more fun than text games. Let's learn how to create games with animated graphics that move.

Animation

In this program we have several different blocks bouncing off of the edges of the window. The blocks are different colors and sizes and move only in diagonal directions. To animate the blocks (that is, make them look like they are moving) we will move the blocks a few pixels over on each iteration through the game loop. This will make it look like the blocks are moving around the screen.

Source Code of the Animation Program

Type the following program into the file editor and save it as *animation.py*. If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/animation.

```
                                                    animation.py
1. import pygame, sys, time
2. from pygame.locals import *
3.
4. # set up pygame
```

```
 5. pygame.init()
 6.
 7. # set up the window
 8. WINDOWWIDTH = 400
 9. WINDOWHEIGHT = 400
10. windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT), 0, 32)
11. pygame.display.set_caption('Animation')
12.
13. # set up direction variables
14. DOWNLEFT = 1
15. DOWNRIGHT = 3
16. UPLEFT = 7
17. UPRIGHT = 9
18.
19. MOVESPEED = 4
20.
21. # set up the colors
22. BLACK = (0, 0, 0)
23. RED = (255, 0, 0)
24. GREEN = (0, 255, 0)
25. BLUE = (0, 0, 255)
26.
27. # set up the block data structure
28. b1 = {'rect':pygame.Rect(300, 80, 50, 100), 'color':RED, 'dir':UPRIGHT}
29. b2 = {'rect':pygame.Rect(200, 200, 20, 20), 'color':GREEN, 'dir':UPLEFT}
30. b3 = {'rect':pygame.Rect(100, 150, 60, 60), 'color':BLUE, 'dir':DOWNLEFT}
31. blocks = [b1, b2, b3]
32.
33. # run the game loop
34. while True:
35.     # check for the QUIT event
36.     for event in pygame.event.get():
37.         if event.type == QUIT:
38.             pygame.quit()
39.             sys.exit()
40.
41.     # draw the black background onto the surface
42.     windowSurface.fill(BLACK)
43.
44.     for b in blocks:
45.         # move the block data structure
46.         if b['dir'] == DOWNLEFT:
47.             b['rect'].left -= MOVESPEED
48.             b['rect'].top += MOVESPEED
49.         if b['dir'] == DOWNRIGHT:
50.             b['rect'].left += MOVESPEED
51.             b['rect'].top += MOVESPEED
```

```
52.             if b['dir'] == UPLEFT:
53.                 b['rect'].left -= MOVESPEED
54.                 b['rect'].top -= MOVESPEED
55.             if b['dir'] == UPRIGHT:
56.                 b['rect'].left += MOVESPEED
57.                 b['rect'].top -= MOVESPEED
58.
59.         # check if the block has move out of the window
60.         if b['rect'].top < 0:
61.             # block has moved past the top
62.             if b['dir'] == UPLEFT:
63.                 b['dir'] = DOWNLEFT
64.             if b['dir'] == UPRIGHT:
65.                 b['dir'] = DOWNRIGHT
66.         if b['rect'].bottom > WINDOWHEIGHT:
67.             # block has moved past the bottom
68.             if b['dir'] == DOWNLEFT:
69.                 b['dir'] = UPLEFT
70.             if b['dir'] == DOWNRIGHT:
71.                 b['dir'] = UPRIGHT
72.         if b['rect'].left < 0:
73.             # block has moved past the left side
74.             if b['dir'] == DOWNLEFT:
75.                 b['dir'] = DOWNRIGHT
76.             if b['dir'] == UPLEFT:
77.                 b['dir'] = UPRIGHT
78.         if b['rect'].right > WINDOWWIDTH:
79.             # block has moved past the right side
80.             if b['dir'] == DOWNRIGHT:
81.                 b['dir'] = DOWNLEFT
82.             if b['dir'] == UPRIGHT:
83.                 b['dir'] = UPLEFT
84.
85.         # draw the block onto the surface
86.         pygame.draw.rect(windowSurface, b['color'], b['rect'])
87.
88.     # draw the window onto the screen
89.     pygame.display.update()
90.     time.sleep(0.02)
```
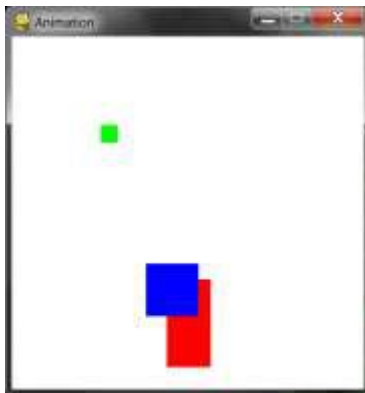
Figure 17-6: An altered screenshot of the Animation program.

How the Animation Program Works

In this program, we will have three different colored blocks moving around and bouncing off the walls. To do this, we need to first consider how we want the blocks to move.

Moving and Bouncing the Blocks

Each block will move in one of four diagonal directions. When the block hits the side of the window, it should bounce off the side and move in a new diagonal direction. The blocks will bounce as shown Figure 17-7.

The new direction that a block moves after it bounces depends on two things: which direction it was moving before the bounce and which wall it bounced off of. There are a total of eight possible ways a block can bounce: two different ways for each of the four walls.

For example, if a block is moving down and right, and then bounces off of the bottom edge of the window, we want the block's new direction to be up and right.

We can represent the blocks with a Rect object to represent the position and size of the block, a tuple of three integers to represent the color of the block, and an integer to represent which of the four diagonal directions the block is currently moving.

On each iteration in the game loop, adjust the X and Y position of the block in the Rect object. Also, in each iteration draw all the blocks on the screen at their current position. As the program execution iterates over the game loop, the blocks will gradually move across the screen so that it looks like they are smoothly moving and bouncing around on their own.
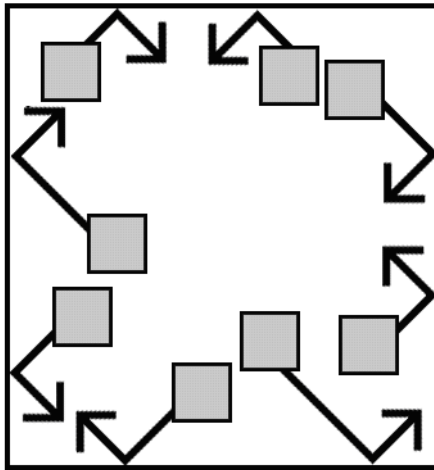
Figure 17-7: The diagram of how blocks will bounce.

Creating and Setting Up Pygame and the Main Window

```
 1. import pygame, sys, time
 2. from pygame.locals import *
 3.
 4. # set up pygame
 5. pygame.init()
 6.
 7. # set up the window
 8. WINDOWWIDTH = 400
 9. WINDOWHEIGHT = 400
10. windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT), 0, 32)
```

In this program, you'll see that the size of the window's width and height is used for more than just the call to `set_mode()`. Use constant variables so that if you ever want to change the size of the window, you only have to change lines 8 and 9. Since the window width and height never change during the program's execution, a constant variable is a good idea.

```
11. pygame.display.set_caption('Animation')
```

Line 11 sets the window's caption to `'Animation'` by calling `pygame.display.set_caption()`.

Setting Up Constant Variables for Direction

```
13. # set up direction variables
14. DOWNLEFT = 1
```

```
15. DOWNRIGHT = 3
16. UPLEFT = 7
17. UPRIGHT = 9
```

We will use the keys on the number pad of the keyboard to remind us which belongs to which direction. This is similar to the Tic Tac Toe game. 1 is down and left, 3 is down and right, 7 is up and left, and 9 is up and right. However, it may be hard to remember this, so instead use constant variables instead of these integer values.

You could have used any value you wanted for these directions instead of using a constant variable. For example, you could use the string `'downleft'` to represent the down and left diagonal direction. However, if you ever mistype the `'downleft'` string (for example, as `'fownleft'`), Python would not recognize that you meant to type `'downleft'` instead of `'downleft'`. This bug would cause your program to behave strangely, but the program would not crash.

But if you use constant variables, and accidentally type the variable name `FOWNLEFT` instead of the name `DOWNLEFT`, Python would notice that there's no such variable named `FOWNLEFT` and crash the program with an error. This would still be a pretty bad bug, but at least you would know about it immediately and could fix it.

```
19. MOVESPEED = 4
```

Use a constant variable to determine how fast the blocks should move. A value of 4 here means that each block will move 4 pixels on each iteration through the game loop.

Setting Up Constant Variables for Color

```
21. # set up the colors
22. BLACK = (0, 0, 0)
23. RED   = (255, 0, 0)
24. GREEN = (0, 255, 0)
25. BLUE  = (0, 0, 255)
```

Lines 22 to 25 set up constant variables for the colors. Remember, Pygame uses a tuple of three integer values for the amounts of red, green, and blue called an RGB value. The integers are from 0 to 255.

The use of constant variables is for readability. The computer doesn't care if you use a variable named GREEN for the color green. It is easier to know that GREEN stands for the color green, rather than (0, 255, 0).

Setting Up The Block Data Structures

```
27. # set up the block data structure
28. b1 = {'rect':pygame.Rect(300, 80, 50, 100), 'color':RED, 'dir':UPRIGHT}
```

Set up a dictionary as a data structure that represents each block. (Chapter 9½ introduced dictionaries.) The dictionary will have the keys of `'rect'` (with a Rect object for a value), `'color'` (with a tuple of three integers for a value), and `'dir'` (with one of the direction constant variables for a value).

The variable `b1` will store one of these block data structures. This block has its top left corner located at an X-coordinate of 300 and Y-coordinate of 80. It has a width of 50 pixels and a height of 100 pixels. Its color is red and its direction is set to UPRIGHT.

```
29. b2 = {'rect':pygame.Rect(200, 200, 20, 20), 'color':GREEN, 'dir':UPLEFT}
30. b3 = {'rect':pygame.Rect(100, 150, 60, 60), 'color':BLUE, 'dir':DOWNLEFT}
```

Line 29 and 30 creates two more similar data structures for blocks that are different sizes, positions, colors, and directions.

```
31. blocks = [b1, b2, b3]
```

Line 31 put all of these data structures in a list, and store the list in a variable named `blocks`.

The `blocks` variable stores a list. `blocks[0]` would be the dictionary data structure in `b1`. `blocks[0]['color']` would be the `'color'` key in `b1`, so the expression `blocks[0]['color']` would evaluate to `(255, 0, 0)`. This way you can refer to any of the values in any of the block data structures by starting with `blocks`.

Running the Game Loop

```
33. # run the game loop
34. while True:
35.     # check for the QUIT event
36.     for event in pygame.event.get():
37.         if event.type == QUIT:
38.             pygame.quit()
39.             sys.exit()
```

Inside the game loop, the blocks will move around the screen in the direction that they are going and bounce if they have hit a side. There is also code to draw all of the blocks to the windowSurface surface and call pygame.display.update().

The for loop to check all of the events in the list returned by pygame.event.get() is the same as in our "Hello World!" program.

```
41.      # draw the black background onto the surface
42.      windowSurface.fill(BLACK)
```

First, line 42 fills the entire surface with black so that anything previously drawn on the surface is erased.

Moving Each Block

```
44.      for b in blocks:
```

Next, the code must update the position of each block, so iterate over the blocks list. Inside the loop, you'll refer to the current block as simply b so it will be easy to type.

```
45.          # move the block data structure
46.          if b['dir'] == DOWNLEFT:
47.              b['rect'].left -= MOVESPEED
48.              b['rect'].top += MOVESPEED
49.          if b['dir'] == DOWNRIGHT:
50.              b['rect'].left += MOVESPEED
51.              b['rect'].top += MOVESPEED
52.          if b['dir'] == UPLEFT:
53.              b['rect'].left -= MOVESPEED
54.              b['rect'].top -= MOVESPEED
55.          if b['dir'] == UPRIGHT:
56.              b['rect'].left += MOVESPEED
57.              b['rect'].top -= MOVESPEED
```

The new value to set the left and top attributes to depends on the block's direction. If the direction of the block (which is stored in the 'dir' key) is either DOWNLEFT or DOWNRIGHT, you want to *increase* the top attribute. If the direction is UPLEFT or UPRIGHT, you want to *decrease* the top attribute.

If the direction of the block is DOWNRIGHT or UPRIGHT, you want to *increase* the left attribute. If the direction is DOWNLEFT or UPLEFT, you want to *decrease* the left attribute.

Change the value of these attributes by the integer stored in MOVESPEED. MOVESPEED stores how many pixels over blocks move on each iteration of the game loop, and was set on line19.

Checking if the Block has Bounced

```
59.          # check if the block has move out of the window
60.          if b['rect'].top < 0:
61.              # block has moved past the top
62.              if b['dir'] == UPLEFT:
63.                  b['dir'] = DOWNLEFT
64.              if b['dir'] == UPRIGHT:
65.                  b['dir'] = DOWNRIGHT
```

After lines 44 to 57 move the block, check if the block has gone past the edge of the window. If it has, you want to "bounce" the block. In the code this means set a new value for the block's 'dir' key. The block will move in the new direction on the next iteration of the game loop. This makes it look like the block has bounced off the side of the window.

On line 60's if statement, the block has moved past the top edge of the window if the block's Rect object's top attribute is less than 0. In that case, change the direction based on what direction the block was moving (either UPLEFT or UPRIGHT).

Changing the Direction of the Bouncing Block

Look at the bouncing diagram earlier in this chapter. To move past the top edge of the window, the block had to either be moving in the UPLEFT or UPRIGHT directions. If the block was moving in the UPLEFT direction, the new direction (according to the bounce diagram) will be DOWNLEFT. If the block was moving in the UPRIGHT direction, the new direction will be DOWNRIGHT.

```
66.          if b['rect'].bottom > WINDOWHEIGHT:
67.              # block has moved past the bottom
68.              if b['dir'] == DOWNLEFT:
69.                  b['dir'] = UPLEFT
70.              if b['dir'] == DOWNRIGHT:
71.                  b['dir'] = UPRIGHT
```

Lines 66 to 71 handles if the block has moved past the bottom edge of the window. They check if the bottom attribute (not the top attribute) is *greater* than the value in WINDOWHEIGHT. Remember that the Y-coordinates start at 0 at the top of the window and increase to WINDOWHEIGHT at the bottom.

The rest of the code changes the direction based on what the bounce diagram in Figure 17-7 says.

```
72.            if b['rect'].left < 0:
73.                # block has moved past the left side
74.                if b['dir'] == DOWNLEFT:
75.                    b['dir'] = DOWNRIGHT
76.                if b['dir'] == UPLEFT:
77.                    b['dir'] = UPRIGHT
78.            if b['rect'].right > WINDOWWIDTH:
79.                # block has moved past the right side
80.                if b['dir'] == DOWNRIGHT:
81.                    b['dir'] = DOWNLEFT
82.                if b['dir'] == UPRIGHT:
83.                    b['dir'] = UPLEFT
```

Lines 78 to 83 are similar to lines 72 to 77, but checks if the left side of the block has moved to the left of the left edge of the window. Remember, the X-coordinates start at 0 on the left edge of the window and increase to WINDOWWIDTH on the right edge of the window.

Drawing the Blocks on the Window in Their New Positions

```
85.        # draw the block onto the surface
86.        pygame.draw.rect(windowSurface, b['color'], b['rect'])
```

Now that the blocks have moved, they should be drawn in their new positions on the windowSurface surface by calling the pygame.draw.rect() function. Pass windowSurface because it is the Surface object to draw the rectangle on. Pass the b['color'] because it is the color of the rectangle. Pass b['rect'] because it is the Rect object with the position and size of the rectangle to draw.

Line 86 is the last line of the for loop. If you wanted to add new blocks, you only have to modify the blocks list on line 31 and the rest of the code still works.

Drawing the Window on the Screen

```
88.    # draw the window onto the screen
89.    pygame.display.update()
90.    time.sleep(0.02)
```

After each of the blocks in the blocks list has been drawn, call pygame.display.update() so that the windowSurface surface is draw on the screen.

After this line, the execution loops back to the start of the game loop and begin the process all over again. This way, the blocks are constantly moving a little, bouncing off the walls, and being drawn on the screen in their new positions.

The call to the `time.sleep()` function is there because the computer can move, bounce, and draw the blocks so fast that if the program ran at full speed, all the blocks would look like a blur. (Try commenting out the `time.sleep(0.02)` line and running the program to see this.)

This call to `time.sleep()` will stop the program for 0.02 seconds, or 20 milliseconds.

Drawing Trails of Blocks

Comment out line 42 (the `windowSurface.fill(BLACK)` line) by adding a # to the front of the line. Now run the program.

Without the call to `windowSurface.fill(BLACK)`, you don't black out the entire window before drawing the rectangles in their new position. The trails of rectangles appear because the old rectangles drawn in previous iterations through the game loop aren't blacked out anymore.

Remember that the blocks are not really moving. On each iteration through the game loop, the code redraws the entire window with new blocks that are located a few pixels over each time.

Summary

This chapter has presented a whole new way of creating computer programs. The previous chapters' programs would stop and wait for the player to enter text. However, in our animation program, the program is constantly updating the data structures of things without waiting for input from the player.

Remember in our Hangman and Tic Tac Toe games we had data structures that would represent the state of the board, and these data structures would be passed to a `drawBoard()` function to be displayed on the screen. Our animation program is similar. The `blocks` variable holds a list of data structures representing blocks to be drawn to the screen, and these are drawn to the screen inside the game loop.

But without calls to `input()`, how do we get input from the player? In our next chapter, we will cover how programs can know when the player presses keys on the keyboard. We will also learn of a concept called collision detection.

Chapter 18

COLLISION DETECTION AND KEYBOARD/MOUSE INPUT

> Topics Covered In This Chapter:
> * Collision Detection
> * Don't Modify a List While Iterating Over It
> * Keyboard Input in Pygame
> * Mouse Input in Pygame

Collision detection is figuring when two things on the screen have touched (that is, collided with) each other. For example, if the player touches an enemy they may lose health. Or the program needs to know when the player touches a coin so that they automatically pick it up. Collision detection can help determine if the game character is standing on solid ground or if there's nothing but empty air underneath them.

In our games, collision detection will determine if two rectangles are overlapping each other or not. Our next example program will cover this basic technique.

Later in this chapter, we'll look at how our Pygame programs can accept input from the user through the keyboard and the mouse. It's a bit more complicated than calling the `input()` function like we did for our text programs. But using the keyboard is much more interactive in GUI programs. And using the mouse isn't even possible in our text games. These two concepts will make your games more exciting!

Source Code of the Collision Detection Program

Much of this code is similar to the animation program, so the explanation of the moving and bouncing code is skipped. (See the animation program in Chapter 17 for that.) A bouncer will bounce around the window. A list of Rect objects will represent food squares.

On each iteration through the game loop, the program will read each Rect object in the list and draw a green square on the window. Every forty iterations through the game loop we will add a new Rect object to the list so that the screen constantly has new food squares in it.

The bouncer is represented by a dictionary. The dictionary has a key named `'rect'` (whose value is a `pygame.Rect` object) and a key named `'dir'` (whose value is one of the constant direction variables like we had in last chapter's Animation program).

As the bouncer bounces around the window, we check if it collides with any of the food squares. If it does, we delete that food square so that it will no longer be drawn on the screen. This will make it look like the bouncer "eats" the food squares in the window.

Type the following into a new file and save it as *collisionDetection.py*. If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/collisionDetection.

```
                                                   collisionDetection.py
 1. import pygame, sys, random
 2. from pygame.locals import *
 3.
 4. def doRectsOverlap(rect1, rect2):
 5.     for a, b in [(rect1, rect2), (rect2, rect1)]:
 6.         # Check if a's corners are inside b
 7.         if ((isPointInsideRect(a.left, a.top, b)) or
 8.             (isPointInsideRect(a.left, a.bottom, b)) or
 9.             (isPointInsideRect(a.right, a.top, b)) or
10.             (isPointInsideRect(a.right, a.bottom, b))):
11.             return True
12.
13.     return False
14.
15. def isPointInsideRect(x, y, rect):
16.     if (x > rect.left) and (x < rect.right) and (y > rect.top) and (y <
rect.bottom):
17.         return True
18.     else:
19.         return False
20.
21.
22. # set up pygame
23. pygame.init()
24. mainClock = pygame.time.Clock()
25.
26. # set up the window
27. WINDOWWIDTH = 400
28. WINDOWHEIGHT = 400
29. windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT), 0,
32)
30. pygame.display.set_caption('Collision Detection')
31.
```

```
32. # set up direction variables
33. DOWNLEFT = 1
34. DOWNRIGHT = 3
35. UPLEFT = 7
36. UPRIGHT = 9
37.
38. MOVESPEED = 4
39.
40. # set up the colors
41. BLACK = (0, 0, 0)
42. GREEN = (0, 255, 0)
43. WHITE = (255, 255, 255)
44.
45. # set up the bouncer and food data structures
46. foodCounter = 0
47. NEWFOOD = 40
48. FOODSIZE = 20
49. bouncer = {'rect':pygame.Rect(300, 100, 50, 50), 'dir':UPLEFT}
50. foods = []
51. for i in range(20):
52.     foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH - FOODSIZE),
random.randint(0, WINDOWHEIGHT - FOODSIZE), FOODSIZE, FOODSIZE))
53.
54. # run the game loop
55. while True:
56.     # check for the QUIT event
57.     for event in pygame.event.get():
58.         if event.type == QUIT:
59.             pygame.quit()
60.             sys.exit()
61.
62.     foodCounter += 1
63.     if foodCounter >= NEWFOOD:
64.         # add new food
65.         foodCounter = 0
66.         foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH -
FOODSIZE), random.randint(0, WINDOWHEIGHT - FOODSIZE), FOODSIZE, FOODSIZE))
67.
68.     # draw the black background onto the surface
69.     windowSurface.fill(BLACK)
70.
71.     # move the bouncer data structure
72.     if bouncer['dir'] == DOWNLEFT:
73.         bouncer['rect'].left -= MOVESPEED
74.         bouncer['rect'].top += MOVESPEED
75.     if bouncer['dir'] == DOWNRIGHT:
76.         bouncer['rect'].left += MOVESPEED
```

```
77.              bouncer['rect'].top += MOVESPEED
78.          if bouncer['dir'] == UPLEFT:
79.              bouncer['rect'].left -= MOVESPEED
80.              bouncer['rect'].top -= MOVESPEED
81.          if bouncer['dir'] == UPRIGHT:
82.              bouncer['rect'].left += MOVESPEED
83.              bouncer['rect'].top -= MOVESPEED
84.
85.          # check if the bouncer has move out of the window
86.          if bouncer['rect'].top < 0:
87.              # bouncer has moved past the top
88.              if bouncer['dir'] == UPLEFT:
89.                  bouncer['dir'] = DOWNLEFT
90.              if bouncer['dir'] == UPRIGHT:
91.                  bouncer['dir'] = DOWNRIGHT
92.          if bouncer['rect'].bottom > WINDOWHEIGHT:
93.              # bouncer has moved past the bottom
94.              if bouncer['dir'] == DOWNLEFT:
95.                  bouncer['dir'] = UPLEFT
96.              if bouncer['dir'] == DOWNRIGHT:
97.                  bouncer['dir'] = UPRIGHT
98.          if bouncer['rect'].left < 0:
99.              # bouncer has moved past the left side
100.             if bouncer['dir'] == DOWNLEFT:
101.                 bouncer['dir'] = DOWNRIGHT
102.             if bouncer['dir'] == UPLEFT:
103.                 bouncer['dir'] = UPRIGHT
104.         if bouncer['rect'].right > WINDOWWIDTH:
105.             # bouncer has moved past the right side
106.             if bouncer['dir'] == DOWNRIGHT:
107.                 bouncer['dir'] = DOWNLEFT
108.             if bouncer['dir'] == UPRIGHT:
109.                 bouncer['dir'] = UPLEFT
110.
111.         # draw the bouncer onto the surface
112.         pygame.draw.rect(windowSurface, WHITE, bouncer['rect'])
113.
114.         # check if the bouncer has intersected with any food squares.
115.         for food in foods[:]:
116.             if doRectsOverlap(bouncer['rect'], food):
117.                 foods.remove(food)
118.
119.         # draw the food
120.         for i in range(len(foods)):
121.             pygame.draw.rect(windowSurface, GREEN, foods[i])
122.
123.         # draw the window onto the screen
```

```
124.       pygame.display.update()
125.       mainClock.tick(40)
```

The program will look like Figure 18-1. The the bouncer square will bounce around the window. When it collides with the green food squares they will disappear from the screen.

Figure 18-1: An altered screenshot of the Collision Detection program.

Importing the Modules

```
1. import pygame, sys, random
2. from pygame.locals import *
```

The collision detection program imports the same things as the Animation program in the last chapter, along with the random module.

The Collision Detection Algorithm

```
4. def doRectsOverlap(rect1, rect2):
```

To do collision detection, you need a function that can determine if two rectangles collide with each other or not. Figure 18-2 shows colliding and non-colliding rectangles.

Figure 18-2: Examples of colliding rectangles (left) and rectangles that don't collide (right).

doRectsOverlap() is passed two pygame.Rect objects. The function will return True if they do and False if they don't. There is a simple rule to follow to determine if rectangles collide. Look at each of the four corners on both rectangles. If at least one of these eight corners is inside the other rectangle, then you know that the two rectangles have collided. You can use this fact to determine if doRectsOverlap() returns True or False.

```
 5.      for a, b in [(rect1, rect2), (rect2, rect1)]:
 6.          # Check if a's corners are inside b
 7.          if ((isPointInsideRect(a.left, a.top, b)) or
 8.              (isPointInsideRect(a.left, a.bottom, b)) or
 9.              (isPointInsideRect(a.right, a.top, b)) or
10.              (isPointInsideRect(a.right, a.bottom, b))):
11.              return True
```

Lines 5 to 11 check if one rectangle's corners are inside another. Later you will create a function called isPointInsideRect() that returns True if the XY coordinates of the point are inside the rectangle. Call this function for each of the eight corners, and if any of these calls return True, the or operators will make the entire condition True.

The parameters for doRectsOverlap() are rect1 and rect2. First check if rect1's corners are inside rect2, then check if rect2's corners are in rect1.

You don't need to repeat the code that checks all four corners for both rect1 and rect2. Instead, use a and b on lines 7 to 10. The for loop on line 5 uses multiple assignment. On the first iteration, a is set to rect1 and b is set to rect2. On the second iteration through the loop, it is the opposite: a is set to rect2 and b is set to rect1.

```
13.      return False
```

Line 11 never returns True, then none of the eight corners checked are in the other rectangle. In that case, the rectangles didn't collide and line 13 returns False.

Determining if a Point is Inside a Rectangle

```
15. def isPointInsideRect(x, y, rect):
16.     if (x > rect.left) and (x < rect.right) and (y > rect.top) and (y <
rect.bottom):
17.         return True
```

The isPointInsideRect() function is called from doRectsOverlap(). The isPointInsideRect() function will return True if the XY coordinates passed are located inside the pygame.Rect object passed as the third parameter. Otherwise, this function returns False.

Figure 18-3 is an example picture of a rectangle and several points. The points and the corners of the rectangle are labeled with coordinates.

A point is inside the rectangle if the following four things are true:

- The point's X-coordinate is greater than the X-coordinate of the rectangle's left side.

- The point's X-coordinate is less than the X-coordinate of the rectangle's right side.

- The point's Y-coordinate is greater than the Y-coordinate of the rectangle's top side.

- The point's Y-coordinate is less than the Y-coordinate of the rectangle's bottom side.

If any of those parts are False, then the point is outside the rectangle. Line 16 combines all four of these conditions into the if statement's condition with and operators.
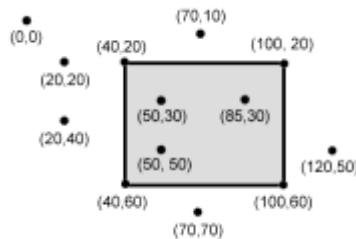

Figure 18-3: Example of coordinates inside and outside of a rectangle. The (50, 30), (85, 30) and (50, 50) points are inside the rectangle, and all the others are outside.

```
18.        else:
19.            return False
```

This function is called from the doRectsOverlap() function to see if any of the corners in the two pygame.Rect objects are inside each other. These two functions give you the power to do collision detection between two rectangles.

The *pygame.time.Clock* Object and *tick()* Method

Much of lines 22 to 43 do the same things that the Animation program in the last chapter did: initialize Pygame, set WINDOWHEIGHT and WINDOWWIDTH, and assign the color and direction constants.

However, line 24 is new:

```
24. mainClock = pygame.time.Clock()
```

In the previous Animation program, a call to `time.sleep(0.02)` would slow down the program so that the program doesn't run too fast. The problem with `time.sleep()` is that might be too much of a pause on slow computers and not enough of a pause on fast computers.

A `pygame.time.Clock` object can pause an appropriate amount of time on any computer. Line 125 calls `mainClock.tick(40)` inside the game loop. This call to the Clock object's `tick()` method waits enough time so that it runs at about 40 iterations a second, no matter what the computer's speed is. This ensures that the game never runs faster than you expect. A call to `tick()` should only appear once in the game loop.

Setting Up the Window and Data Structures

```
45. # set up the bouncer and food data structures
46. foodCounter = 0
47. NEWFOOD = 40
48. FOODSIZE = 20
```

Lines 46 to 48 set up a few variables for the food blocks that appear on the screen. `foodCounter` will start at the value 0, `NEWFOOD` at 40, and `FOODSIZE` at 20.

```
49. bouncer = {'rect':pygame.Rect(300, 100, 50, 50), 'dir':UPLEFT}
```

Line 49 sets up a new data structure called `bouncer`. `bouncer` is a dictionary with two keys. The `'rect'` key has a `pygame.Rect` object that represents the bouncer's size and position.

The `'dir'` key has a direction that the bouncer is currently moving. The bouncer will move the same way the blocks did in Chapter 17's animation program.

```
50. foods = []
51. for i in range(20):
52.     foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH - FOODSIZE),
random.randint(0, WINDOWHEIGHT - FOODSIZE), FOODSIZE, FOODSIZE))
```

The program will keep track of every food square with a list of Rect objects in `foods`. Lines 51 and 52 create twenty food squares randomly placed around the screen. You can use the `random.randint()` function to come up with random XY coordinates.

On line 52, we will call the `pygame.Rect()` constructor function to return a new `pygame.Rect` object. It will represent the position and size of the food square. The first two parameters for `pygame.Rect()` are the XY coordinates of the top left corner. You want the random coordinate to be between 0 and the size of the window minus the size of the food square. If you had the random

coordinate between 0 and the size of the window, then the food square might be pushed outside of the window altogether, like in Figure 18-4.
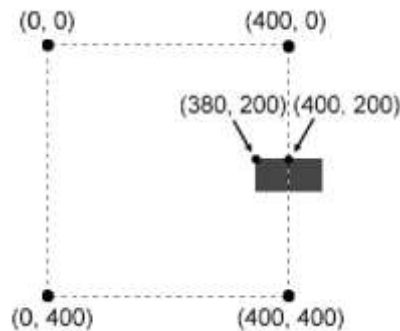

Figure 18-4: For a 20 by 20 rectangle, having the top left corner at (400, 200) in a 400 by 400 window would place the rectangle outside of the window. To be inside, the top left corner should be at (380, 200) instead.

The third parameter for `pygame.Rect()` is a tuple that contains the width and height of the food square. Both the width and height is the value in the `FOODSIZE` constant.

Drawing the Bouncer on the Screen

Lines 71 to 109 cause the bouncer to move around the window and bounce off of the edges of the window. This code is similar to lines 44 to 83 of the Animation program in the last chapter, so the explanation will be skipped.

```
111.      # draw the bouncer onto the surface
112.      pygame.draw.rect(windowSurface, WHITE, bouncer['rect'])
```

After moving the bouncer, line 112 draws it in its new position. The `windowSurface` passed for the first parameter tells Python which Surface object to draw the rectangle on. The `WHITE` variable, which has `(255, 255, 255)` stored in it, will tell Python to draw a white rectangle. The Rect object stored in the `bouncer` dictionary at the `'rect'` key tells the position and size of the rectangle to draw.

Colliding with the Food Squares

```
114.      # check if the bouncer has intersected with any food squares.
115.      for food in foods[:]:
```

Before drawing the food squares, check if the bouncer has overlapped any of the food squares. If it has, remove that food square from the `foods` list. This way, Python won't draw any food squares that the bouncer has "eaten".

On each iteration through the `for` loop, the current food square from the `foods` (plural) list is in the variable `food` (singular).

Don't Add to or Delete from a List while Iterating Over It

Notice that there's a slight difference with this `for` loop. If you look carefully at line 116, it isn't iterating over `foods` but actually over `foods[:]`.

Remember how slices work. `foods[:2]` evaluates to a copy of the list with the items from the start and up to (but not including) the item at index 2. `foods[3:]` evaluates to a copy of the list with the items from index 3 to the end of the list.

`foods[:]` will give you a copy of the list with the items from the start to the end. Basically, `foods[:]` creates a new list with a copy of all the items in `foods`. This is a shorter way to copy a list than, say, what the `getBoardCopy()` function does in the previous Tic Tac Toe game.

You cannot add or remove items from a list while you are iterating over it. Python can lose track of what the next value of `food` variable should be if the size of the `foods` list is always changing. Think of how difficult it would be to count the number of jelly beans in a jar while someone was adding or removing jelly beans.

But if you iterate over a copy of the list (and the copy never changes), adding or removing items from the original list won't be a problem.

Removing the Food Squares

```
116.          if doRectsOverlap(bouncer['rect'], food):
117.              foods.remove(food)
```

Line 116 is where `doRectsOverlap()` comes in handy. If the bouncer and the current food square two rectangles overlap, then `doRectsOverlap()` will return `True` and line 117 removes the overlapping food square from the `foods` list.

Drawing the Food Squares on the Screen

```
119.     # draw the food
120.     for i in range(len(foods)):
121.         pygame.draw.rect(windowSurface, GREEN, foods[i])
```

The code on lines 120 and 121 are similar to how we drew the white square for the player. Line 120 loops through each food square in the `foods` list. Line 121 draws the food square onto the `windowSurface` surface. This program was similar to the bouncing program in the previous chapter, except now the bouncing square will "eat" other squares it passes over them.

These past few programs are interesting to watch, but the user doesn't get to control anything. In the next program, we will learn how to get input from the keyboard.

Source Code of the Keyboard Input Program

Start a new file and type in the following code, then save it as *pygameInput.py*. If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/pygameInput.

```
                                                            pygameInput.py
 1. import pygame, sys, random
 2. from pygame.locals import *
 3.
 4. # set up pygame
 5. pygame.init()
 6. mainClock = pygame.time.Clock()
 7.
 8. # set up the window
 9. WINDOWWIDTH = 400
10. WINDOWHEIGHT = 400
11. windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT), 0,
32)
12. pygame.display.set_caption('Input')
13.
14. # set up the colors
15. BLACK = (0, 0, 0)
16. GREEN = (0, 255, 0)
17. WHITE = (255, 255, 255)
18.
19. # set up the player and food data structure
20. foodCounter = 0
21. NEWFOOD = 40
22. FOODSIZE = 20
23. player = pygame.Rect(300, 100, 50, 50)
24. foods = []
25. for i in range(20):
26.     foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH - FOODSIZE),
random.randint(0, WINDOWHEIGHT - FOODSIZE), FOODSIZE, FOODSIZE))
27.
28. # set up movement variables
```

```
29. moveLeft = False
30. moveRight = False
31. moveUp = False
32. moveDown = False
33.
34. MOVESPEED = 6
35.
36.
37. # run the game loop
38. while True:
39.     # check for events
40.     for event in pygame.event.get():
41.         if event.type == QUIT:
42.             pygame.quit()
43.             sys.exit()
44.         if event.type == KEYDOWN:
45.             # change the keyboard variables
46.             if event.key == K_LEFT or event.key == ord('a'):
47.                 moveRight = False
48.                 moveLeft = True
49.             if event.key == K_RIGHT or event.key == ord('d'):
50.                 moveLeft = False
51.                 moveRight = True
52.             if event.key == K_UP or event.key == ord('w'):
53.                 moveDown = False
54.                 moveUp = True
55.             if event.key == K_DOWN or event.key == ord('s'):
56.                 moveUp = False
57.                 moveDown = True
58.         if event.type == KEYUP:
59.             if event.key == K_ESCAPE:
60.                 pygame.quit()
61.                 sys.exit()
62.             if event.key == K_LEFT or event.key == ord('a'):
63.                 moveLeft = False
64.             if event.key == K_RIGHT or event.key == ord('d'):
65.                 moveRight = False
66.             if event.key == K_UP or event.key == ord('w'):
67.                 moveUp = False
68.             if event.key == K_DOWN or event.key == ord('s'):
69.                 moveDown = False
70.             if event.key == ord('x'):
71.                 player.top = random.randint(0, WINDOWHEIGHT -
player.height)
72.                 player.left = random.randint(0, WINDOWWIDTH -
player.width)
73.
```

```
74.            if event.type == MOUSEBUTTONUP:
75.                foods.append(pygame.Rect(event.pos[0], event.pos[1], FOODSIZE,
FOODSIZE))
76.
77.        foodCounter += 1
78.        if foodCounter >= NEWFOOD:
79.            # add new food
80.            foodCounter = 0
81.            foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH -
FOODSIZE), random.randint(0, WINDOWHEIGHT - FOODSIZE), FOODSIZE, FOODSIZE))
82.
83.        # draw the black background onto the surface
84.        windowSurface.fill(BLACK)
85.
86.        # move the player
87.        if moveDown and player.bottom < WINDOWHEIGHT:
88.            player.top += MOVESPEED
89.        if moveUp and player.top > 0:
90.            player.top -= MOVESPEED
91.        if moveLeft and player.left > 0:
92.            player.left -= MOVESPEED
93.        if moveRight and player.right < WINDOWWIDTH:
94.            player.right += MOVESPEED
95.
96.        # draw the player onto the surface
97.        pygame.draw.rect(windowSurface, WHITE, player)
98.
99.        # check if the player has intersected with any food squares.
100.        for food in foods[:]:
101.            if player.colliderect(food):
102.                foods.remove(food)
103.
104.        # draw the food
105.        for i in range(len(foods)):
106.            pygame.draw.rect(windowSurface, GREEN, foods[i])
107.
108.        # draw the window onto the screen
109.        pygame.display.update()
110.        mainClock.tick(40)
```

This program is almost identical to the collision detection program. But in this program, the bouncer only moves around when the user holds down the arrow keys on the keyboard.

You can also click anywhere in the window and create new food objects. In addition, the ESC key will quit the program and the "X" key will teleport the player to a random place on the screen.

Setting Up the Window and Data Structures

Starting at line 29, the code sets up some variables that track the movement of the bouncer.

```
28. # set up movement variables
29. moveLeft = False
30. moveRight = False
31. moveUp = False
32. moveDown = False
```

The four variables have Boolean values to keep track of which of the arrow keys are being held down. For example, when the user pushes the left arrow key on their keyboard, `moveLeft` is set to `True`. When they let go of the key, `moveLeft` is set back to `False`.

Lines 34 to 43 are identical to code in the previous Pygame programs. These lines handle the start of the game loop and what to do when the user quits the program. We'll skip the explanation for this code here since we have already covered it in the last chapter.

Events and Handling the KEYDOWN Event

The code to handle the key press and key release events start on line 44. At the start of the program, they are all set to `False`.

```
44.         if event.type == KEYDOWN:
```

Pygame has an event type called `KEYDOWN`. This is one of the other events that Pygame can generate. A brief list of the events that could be returned by `pygame.event.get()` is in Table 18-1.

Table 18-1: Events and when they are generated.

Event Type	Description
QUIT	Generated when the user closes the window.
KEYDOWN	Generated when the user presses down a key. Has a key attribute that tells which key was pressed. Also has a mod attribute that tells if the Shift, Ctrl, Alt, or other keys were held down when this key was pressed.
KEYUP	Generated when the user releases a key. Has a key and mod attribute that are similar to those for KEYDOWN.
MOUSEMOTION	Generated whenever the mouse moves over the window. Has a pos attribute that returns tuple (x, y) for the coordinates of where the mouse is in the window. The rel attribute also returns a (x, y) tuple, but it gives coordinates relative since the last MOUSEMOTION event. For example, if the mouse moves left by four pixels from (200, 200) to (196, 200), then rel will be the tuple value (-4, 0).
	The buttons attribute returns a tuple of three integers. The first integer in the tuple is for the left mouse button, the second integer for the middle mouse button (if there's a middle mouse button), and the third integer is for the right mouse button. These integers will be 0 if they are not being pressed down when the mouse moved and 1 if they are pressed down.
MOUSEBUTTONDOWN	Generated when a mouse button is pressed down in the window. This event has a pos attribute which is an (x, y) tuple for the coordinates of where the mouse was when the button was pressed. There is also a button attribute which is an integer from 1 to 5 that tells which mouse button was pressed, explained in Table 18-2.
MOUSEBUTTONUP	Generated when the mouse button is released. This has the same attributes as MOUSEBUTTONDOWN.

Table 18-2: The `button` attribute values and mouse button.

Value of button	Mouse Button
1	Left button
2	Middle button
3	Right button
4	Scroll wheel moved up
5	Scroll wheel moved down

Setting the Four Keyboard Variables

```
45.                 # change the keyboard variables
46.                 if event.key == K_LEFT or event.key == ord('a'):
47.                     moveRight = False
48.                     moveLeft = True
49.                 if event.key == K_RIGHT or event.key == ord('d'):
50.                     moveLeft = False
51.                     moveRight = True
52.                 if event.key == K_UP or event.key == ord('w'):
53.                     moveDown = False
54.                     moveUp = True
55.                 if event.key == K_DOWN or event.key == ord('s'):
56.                     moveUp = False
57.                     moveDown = True
```

If the event type is KEYDOWN, then the event object will have a key attribute that tells which key was pressed down. Line 46 compares this attribute to K_LEFT, which is the pygame.locals constant that represents the left arrow key on the keyboard. Lines 46 to 57 do similar checks for each of the other arrow keys: K_LEFT, K_RIGHT, K_UP, K_DOWN.

When one of these keys is pressed down, set the corresponding movement variable to True. Also, set the movement variable of the opposite direction to False.

For example, the program executes lines 47 and 48 when the left arrow key has been pressed. In this case, set moveLeft to True and moveRight to False (even though moveRight might already be False, set it to False just to be sure).

On line 46, in event.key can either be equal to K_LEFT or ord('a'). The value in event.key is set to the integer ordinal value of the key that was pressed on the keyboard. (There is no ordinal

value for the arrow keys, which is why we use the constant variable K_LEFT.) You can use the ord() function to get the ordinal value of any single character to compare it with event.key.

By executing the code on lines 47 and 48 if the keystroke was either K_LEFT or ord('a'), you make the left arrow key and the A key do the same thing. The W, A, S, and D keys are all used as alternates for changing the movement variables. The WASD (pronounced "wazz-dee") keys let you use your left hand. The arrow keys can be pressed with your right hand.

Figure 18-5: The WASD keys can be programmed to do the same thing as the arrow keys.

Handling the KEYUP Event

```
58.            if event.type == KEYUP:
```

When the user releases the key that they are holding down, a KEYUP event is generated.

```
59.                if event.key == K_ESCAPE:
60.                    pygame.quit()
61.                    sys.exit()
```

If the key that the user released was the ESC key, then terminate the program. Remember, in Pygame you must call the pygame.quit() function before calling the sys.exit() function.

Lines 62 to 69 will set a movement variable to False if that direction's key was let go.

```
62.                if event.key == K_LEFT or event.key == ord('a'):
63.                    moveLeft = False
64.                if event.key == K_RIGHT or event.key == ord('d'):
65.                    moveRight = False
66.                if event.key == K_UP or event.key == ord('w'):
67.                    moveUp = False
68.                if event.key == K_DOWN or event.key == ord('s'):
69.                    moveDown = False
```

Teleporting the Player

```
70.                if event.key == ord('x'):
```

```
71.                     player.top = random.randint(0, WINDOWHEIGHT -
player.height)
72.                     player.left = random.randint(0, WINDOWWIDTH -
player.width)
```

You can also add teleportation to the game. If the user presses the "X" key, then lines 71 and 72 will set the position of the user's square to a random place on the window. This will give the user the ability to teleport around the window by pushing the "X" key. Although they can't control where they will teleport; it's completely random.

Handling the MOUSEBUTTONUP Event

```
74.             if event.type == MOUSEBUTTONUP:
75.                 foods.append(pygame.Rect(event.pos[0], event.pos[1], FOODSIZE,
FOODSIZE))
```

Mouse input is handled by events just like keyboard input is. The MOUSEBUTTONUP event occurs when the user releases the mouse button after clicking it. The pos attribute in the Event object is set to a tuple of two integers for the XY coordinates for where the mouse cursor was at the time of the click.

On line 75, the X-coordinate is stored in event.pos[0] and the Y-coordinate is stored in event.pos[1]. Line 75 creates a new Rect object to represent a new food and place it where the MOUSEBUTTONUP event occurred. By adding a new Rect object to the foods list, the code will display a new food square is displayed on the screen.

Moving the Player Around the Screen

```
86.      # move the player
87.      if moveDown and player.bottom < WINDOWHEIGHT:
88.          player.top += MOVESPEED
89.      if moveUp and player.top > 0:
90.          player.top -= MOVESPEED
91.      if moveLeft and player.left > 0:
92.          player.left -= MOVESPEED
93.      if moveRight and player.right < WINDOWWIDTH:
94.          player.right += MOVESPEED
```

You've set the movement variables (moveDown, moveUp, moveLeft, and moveRight) to True or False depending on what keys the user has pressed. Now move the player's square (which is represented by the pygame.Rect object stored in player) by adjusting XY coordinates of player.

If moveDown is set to True (and the bottom of the player's square isn't below the bottom edge of the window), then line 88 moves the player's square down by adding MOVESPEED to the player's current top attribute. Lines 89 to 94 do the same thing for the other three directions.

The colliderect() Method

```
 99.        # check if the player has intersected with any food squares.
100.        for food in foods[:]:
101.            if player.colliderect(food):
102.                foods.remove(food)
```

In the previous Collision Detection program, the doRectsOverlap() function to check if one rectangle had collided with another. That function was included in this book so you could understand how the code behind collision detection works.

In this program, you can use the collision detection function that comes with Pygame. The colliderect() method for pygame.Rect objects is passed another pygame.Rect object as an argument and returns True if the two rectangles collide and False if they do not.

```
110.        mainClock.tick(40)
```

The rest of the code is similar to the code in the Input and Collision Detection programs.

Summary

This chapter introduced the concept of collision detection, which is in many graphical games. Detecting collisions between two rectangles is easy: check if the four corners of either rectangle are within the other rectangle. This is such a common thing to check for that Pygame provides its own collision detection method named colliderect() for pygame.Rect objects.

The first several games in this book were text-based. The program output was text printed to the screen and the input was text typed by the user on the keyboard. But graphical programs can accept keyboard and mouse inputs.

Furthermore, these programs can respond to single keystrokes when the user pushes down or lets up a single key. The user doesn't have to type in an entire response and press ENTER. This allows for immediate feedback and much more interactive games.

Chapter 19

SOUNDS AND IMAGES

> Topics Covered In This Chapter:
> - Sound and Image Files
> - Drawing Sprites
> - The `pygame.image.load()` Function
> - The `pygame.mixer.Sound` Data Type
> - The `pygame.mixer.music` Module

In the last two chapters, we've learned how to make GUI programs that have graphics and can accept input from the keyboard and mouse. We've also learned how to draw different shapes. In this chapter, we will learn how to show pictures and images (called sprites) and play sounds and music in our games.

A **sprite** is a name for a single two-dimensional image that is used as part of the graphics on the screen. Figure 19-1 shows some example sprites.

Figure 19-1: Some examples of sprites.

Figure 19-2 shows being used in a complete scene.

Figure 19-2: An example of a complete scene, with sprites drawn on top of a background.

The sprite images are drawn on top of a background. Notice that you can flip the sprite image horizontally so that the sprites are facing the other way. You can draw the same sprite image multiple times on the same window. You can also resize the sprites to be larger or smaller than the original sprite image. The background image can be considered one large sprite.

The next program will demonstrate how to play sounds and draw sprites using Pygame.

Sound and Image Files

Sprites are stored in image files on your computer. There are several different image formats that Pygame can use. You can tell what format an image file uses by looking at the end of the file name (after the last period). This is called the **file extension**. For example, the file *player.png* is in the PNG format. The image formats Pygame supports include BMP, PNG, JPG, and GIF.

You can download images from your web browser. On most web browsers, you have to right-click on the image in the web page and select Save from the menu that appears. Remember where on the hard drive you saved the image file. Copy this downloaded image file into the same folder as you Python program's *.py* file. You can also create your own images with a drawing program like MS Paint or Tux Paint.

The sound file formats that Pygame supports are MID, WAV, and MP3. You can download sound effects from the Internet just like image files. They must be in one of these three formats. If

your computer has a microphone, you can also record sounds and make your own WAV files to use in your games.

Sprites and Sounds Program

This program is the same as the Keyboard and Mouse Input program from the last chapter. However, in this program we will use sprites instead of plain looking squares. We will use a sprite of a little person instead of the white player square, and a sprite of cherries instead of the green food squares. We also play background music and a sound effect when the player sprite eats one of the cherry sprites.

Source Code of the Sprites and Sounds Program

If you know how to use graphics software such as Photoshop or MS Paint, you can draw your own images. If you don't know how to use these programs, you can download graphics from websites and use those image files instead. The same applies for music and sound files. You can also find images on websites or images from a digital camera. You can download the image and sound files from this book's website at http://invpy.com/downloads.

If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/spritesAndSounds.

spritesAndSounds.py

```
 1. import pygame, sys, time, random
 2. from pygame.locals import *
 3.
 4. # set up pygame
 5. pygame.init()
 6. mainClock = pygame.time.Clock()
 7.
 8. # set up the window
 9. WINDOWWIDTH = 400
10. WINDOWHEIGHT = 400
11. windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT), 0,
32)
12. pygame.display.set_caption('Sprites and Sound')
13.
14. # set up the colors
15. BLACK = (0, 0, 0)
16.
17. # set up the block data structure
18. player = pygame.Rect(300, 100, 40, 40)
19. playerImage = pygame.image.load('player.png')
```

```
20. playerStretchedImage = pygame.transform.scale(playerImage, (40, 40))
21. foodImage = pygame.image.load('cherry.png')
22. foods = []
23. for i in range(20):
24.     foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH - 20),
random.randint(0, WINDOWHEIGHT - 20), 20, 20))
25.
26. foodCounter = 0
27. NEWFOOD = 40
28.
29. # set up keyboard variables
30. moveLeft = False
31. moveRight = False
32. moveUp = False
33. moveDown = False
34.
35. MOVESPEED = 6
36.
37. # set up music
38. pickUpSound = pygame.mixer.Sound('pickup.wav')
39. pygame.mixer.music.load('background.mid')
40. pygame.mixer.music.play(-1, 0.0)
41. musicPlaying = True
42.
43. # run the game loop
44. while True:
45.     # check for the QUIT event
46.     for event in pygame.event.get():
47.         if event.type == QUIT:
48.             pygame.quit()
49.             sys.exit()
50.         if event.type == KEYDOWN:
51.             # change the keyboard variables
52.             if event.key == K_LEFT or event.key == ord('a'):
53.                 moveRight = False
54.                 moveLeft = True
55.             if event.key == K_RIGHT or event.key == ord('d'):
56.                 moveLeft = False
57.                 moveRight = True
58.             if event.key == K_UP or event.key == ord('w'):
59.                 moveDown = False
60.                 moveUp = True
61.             if event.key == K_DOWN or event.key == ord('s'):
62.                 moveUp = False
63.                 moveDown = True
64.         if event.type == KEYUP:
65.             if event.key == K_ESCAPE:
```

```
 66.                    pygame.quit()
 67.                    sys.exit()
 68.               if event.key == K_LEFT or event.key == ord('a'):
 69.                   moveLeft = False
 70.               if event.key == K_RIGHT or event.key == ord('d'):
 71.                   moveRight = False
 72.               if event.key == K_UP or event.key == ord('w'):
 73.                   moveUp = False
 74.               if event.key == K_DOWN or event.key == ord('s'):
 75.                   moveDown = False
 76.               if event.key == ord('x'):
 77.                   player.top = random.randint(0, WINDOWHEIGHT -
player.height)
 78.                   player.left = random.randint(0, WINDOWWIDTH -
player.width)
 79.               if event.key == ord('m'):
 80.                   if musicPlaying:
 81.                       pygame.mixer.music.stop()
 82.                   else:
 83.                       pygame.mixer.music.play(-1, 0.0)
 84.                   musicPlaying = not musicPlaying
 85.
 86.           if event.type == MOUSEBUTTONUP:
 87.               foods.append(pygame.Rect(event.pos[0] - 10, event.pos[1] - 10,
20, 20))
 88.
 89.       foodCounter += 1
 90.       if foodCounter >= NEWFOOD:
 91.           # add new food
 92.           foodCounter = 0
 93.           foods.append(pygame.Rect(random.randint(0, WINDOWWIDTH - 20),
random.randint(0, WINDOWHEIGHT - 20), 20, 20))
 94.
 95.       # draw the black background onto the surface
 96.       windowSurface.fill(BLACK)
 97.
 98.       # move the player
 99.       if moveDown and player.bottom < WINDOWHEIGHT:
100.           player.top += MOVESPEED
101.       if moveUp and player.top > 0:
102.           player.top -= MOVESPEED
103.       if moveLeft and player.left > 0:
104.           player.left -= MOVESPEED
105.       if moveRight and player.right < WINDOWWIDTH:
106.           player.right += MOVESPEED
107.
108.
```

```
109.      # draw the block onto the surface
110.      windowSurface.blit(playerStretchedImage, player)
111.
112.      # check if the block has intersected with any food squares.
113.      for food in foods[:]:
114.          if player.colliderect(food):
115.              foods.remove(food)
116.              player = pygame.Rect(player.left, player.top, player.width +
2, player.height + 2)
117.              playerStretchedImage = pygame.transform.scale(playerImage,
(player.width, player.height))
118.              if musicPlaying:
119.                  pickUpSound.play()
120.
121.      # draw the food
122.      for food in foods:
123.          windowSurface.blit(foodImage, food)
124.
125.      # draw the window onto the screen
126.      pygame.display.update()
127.      mainClock.tick(40)
```


Figure 19-3: An altered screenshot of the Sprites and Sounds game.

Setting Up the Window and the Data Structure

Most of the code in this program is the same as the Collision Detection program in the previous chapter. We'll focus only on the parts that add sprites and sound.

```
12. pygame.display.set_caption('Sprites and Sound')
```

First, let's set the caption of the title bar to a string that describes this program on line 12. Pass the string `'Sprites and Sound'` to the `pygame.display.set_caption()` function.

```
17. # set up the block data structure
18. player = pygame.Rect(300, 100, 40, 40)
19. playerImage = pygame.image.load('player.png')
20. playerStretchedImage = pygame.transform.scale(playerImage, (40, 40))
21. foodImage = pygame.image.load('cherry.png')
```

We are going to use three different variables to represent the player, unlike the previous programs that just used one.

The `player` variable on line 18 will store a Rect object that keeps track of where and how big the player is. The `player` variable doesn't contain the player's image, only the player's size and location. At the beginning of the program, the top left corner of the player is located at (300, 100) and the player will have a height and width of 40 pixels to start.

The second variable on line 19 that represents the player is `playerImage`. The `pygame.image.load()` function is passed a string of the filename of the image to load. The return value is a Surface object that has the graphics in the image file drawn on its surface. We store this Surface object inside of `playerImage`.

The third variable is explained in the next section.

The `pygame.transform.scale()` Function

On line 20, we will use a new function in the `pygame.transform` module. The `pygame.transform.scale()` function can shrink or enlarge a sprite. The first argument is a `pygame.Surface` object with the image drawn on it. The second argument is a tuple for the new width and height of the image in the first argument. The `pygame.transform.scale()` function returns a `pygame.Surface` object with the image drawn at a new size. We will store the original image in the `playerImage` variable but the stretched image in the `playerStretchedImage` variable.

On line 21, we call `pygame.image.load()` again to create a Surface object with the cherry image drawn on it. Be sure you have the *player.png* and *cherry.png* files in the same directory as the *spritesAndSounds.py* file, otherwise Pygame won't be able to find them and will give an error.

Setting Up the Music and Sounds

```
37. # set up music
38. pickUpSound = pygame.mixer.Sound('pickup.wav')
39. pygame.mixer.music.load('background.mid')
```

```
40. pygame.mixer.music.play(-1, 0.0)
41. musicPlaying = True
```

Next you need to load the sound files. There are two modules for sound in Pygame. The `pygame.mixer` module can play short sound effects during the game. The `pygame.mixer.music` module can play background music.

Call the `pygame.mixer.Sound()` constructor function to create a `pygame.mixer.Sound` object (called a Sound object for short). This object has a `play()` method that when called will play the sound effect when called.

Line 39 calls `pygame.mixer.music.load()` to load the background music. Line 40 calls `pygame.mixer.music.play()` to start playing the background music. The first parameter tells Pygame how many times to play the background music after the first time we play it. So passing `5` will cause Pygame to play the background music 6 times. `-1` is a special value, and passing it for the first parameter makes the background music repeat forever.

The second parameter to `pygame.mixer.music.play()` is the point in the sound file to start playing. Passing `0.0` will play the background music starting from the beginning. Passing `2.5` for the second parameter will start the background music two and half seconds from the beginning.

Finally, the `musicPlaying` variable will have a Boolean value that tells the program if it should play the background music and sound effects or not. It's nice to give the player the option to run the program without the sound playing.

Toggling the Sound On and Off

```
79.                if event.key == ord('m'):
80.                    if musicPlaying:
81.                        pygame.mixer.music.stop()
82.                    else:
83.                        pygame.mixer.music.play(-1, 0.0)
84.                    musicPlaying = not musicPlaying
```

The M key will turn the background music on or off. If `musicPlaying` is set to `True`, then the background music is currently playing and we should stop the music by calling `pygame.mixer.music.stop()`. If `musicPlaying` is set to `False`, then the background music isn't currently playing and should be started by calling `pygame.mixer.music.play()`.

Finally, no matter what, we want to toggle the value in `musicPlaying`. **Toggling** a Boolean value means to set to the opposite of its current value. The line `musicPlaying = not musicPlaying` sets the variable to `False` if it is currently `True` or sets it to `True` if it is currently `False`. Think of

toggling as what happens when you flip a light switch on or off: toggling the light switch sets it to the opposite setting.

Drawing the Player on the Window

```
109.      # draw the block onto the surface
110.      windowSurface.blit(playerStretchedImage, player)
```

Remember that the value stored in `playerStretchedImage` is a Surface object. Line 110 draws the sprite of the player onto the window's Surface object (which is stored in `windowSurface`).

The second parameter to the `blit()` method is a Rect object that specifies where on the Surface object the sprite should be blitted. The Rect object stored in `player` is what keeps track of the position of the player in the window.

Checking if the Player Has Collided with Cherries

```
114.          if player.colliderect(food):
115.              foods.remove(food)
116.              player = pygame.Rect(player.left, player.top, player.width +
2, player.height + 2)
117.              playerStretchedImage = pygame.transform.scale(playerImage,
(player.width, player.height))
118.              if musicPlaying:
119.                  pickUpSound.play()
```

This code is similar to the code in the previous programs. But there are a couple of new lines. Call the `play()` method on the Sound object stored in the `pickUpSound` variable. But only do this if `musicPlaying` is set to `True` (which means that the sound is turned on).

When the player eats one of the cherries, the size of the player increases by two pixels in height and width. On line 116, a new Rect object that is 2 pixels larger than the old Rect object will be the new value of `player`.

While the Rect object represents the position and size of the player, the image of the player is stored in a `playerStretchedImage` as a Surface object. Create a new stretched image by calling `pygame.transform.scale()`. Be sure to pass the original Surface object in `playerImage` and not `playerStretchedImage`.

Stretching an image often distorts it a little. If you keep restretching a stretched image over and over, the distortions add up quickly. But by stretching the original image to the new size, you only distort the image once. This is why you pass `playerImage` as the first argument for `pygame.transform.scale()`.

Draw the Cherries on the Window

```
121.       # draw the food
122.       for food in foods:
123.           windowSurface.blit(foodImage, food)
```

In the previous programs, you called the pygame.draw.rect() function to draw a green square for each Rect object stored in the foods list. However, in this program you want to draw the cherry sprites instead. Call the blit() method and pass the Surface object stored in foodImage. (This is the Surface object with the image of cherries drawn on it.)

The food variable (which contains each of the Rect objects in foods on each iteration through the for loop) tells the blit() method where to draw the foodImage.

Summary

This game has added images and sound to your games. The images (called sprites) look much better than the simple shape drawing used in the previous programs. The game presented in this chapter also has music playing in the background while also playing sound effects.

Sprites can be scaled (that is, stretched) to a larger or smaller size. This way we can display sprites at any size we want. This will come in handy in the game presented in the next chapter.

Now that we know how to create a window, display sprites and drawing primitives, collect keyboard and mouse input, play sounds, and implement collision detection, we are now ready to create a graphical game in Pygame. The next chapter brings all of these elements together for our most advanced game yet.

Chapter 20

DODGER

Topics Covered In This Chapter:
- The `pygame.FULLSCREEN` flag
- Pygame constant variables for keyboard keys
- The `move_ip()`Rect method
- The `pygame.mouse.set_pos()` function
- Implementing cheat codes
- Modifying the Dodger game

The last three chapters went over the Pygame module and demonstrated how to use its many features. In this chapter, we'll use that knowledge to create a graphical game called Dodger.

The Dodger game has the player control a small person (which we call the player's character) who must dodge a whole bunch of baddies that fall from the top of the screen. The longer the player can keep dodging the baddies, the higher the score they will get.

Just for fun, we'll also add some cheat modes to the game. If the player holds down the "x" key, every baddie's speed is reduced to a super slow rate. If the player holds down the "z" key, the baddies will reverse their direction and travel up the screen instead of downwards.

Review of the Basic Pygame Data Types

Let's review some of the basic data types used in Pygame:

- `pygame.Rect` - Rect objects represent a rectangular space's location and size. The location can be determined by the Rect object's `topleft` attribute (or the `topright`, `bottomleft`, and `bottomright` attributes). These corner attributes are a tuple of integers for the X- and Y-coordinates. The size can be determined by the `width` and `height` attributes, which are integers of how many pixels long or high the rectangle area is. Rect objects have a `colliderect()` method to check if they are colliding with another Rect object.

- `pygame.Surface` - Surface objects are areas of colored pixels. Surface objects represent a rectangular image, while Rect objects only represent a rectangular space and location. Surface objects have a `blit()` method that is used to draw the image on one Surface object onto another Surface object. The Surface object returned by the

pygame.display.set_mode() function is special because anything drawn on that Surface object is displayed on the user's screen when pygame.display.update() is called.

- pygame.event.Event - The pygame.event module generates Event objects whenever the user provides keyboard, mouse, or another kind of input. The pygame.event.get() function returns a list of these Event objects. You can check what type of event the Event object is by checking its type attribute. QUIT, KEYDOWN, and MOUSEBUTTONUP are examples of some event types.
- pygame.font.Font - The pygame.font module has the Font data type which represents the typeface used for text in Pygame. The arguments to pass to pygame.font.SysFont() are a string of the font name and an integer of the font size. However it is common to pass None for the font name to get the default system font.
- pygame.time.Clock - The Clock object in the pygame.time module is helpful for keeping our games from running as fast as possible. The Clock object has a tick() method, which we pass how many frames per second (FPS) we want the game to run at. The higher the FPS, the faster the game runs.

Type in the following code and save it to a file named *dodger.py*. This game also requires some other image and sound files, which you can download from the URL http://invpy.com/downloads.

Source Code of Dodger

You can download this code from the URL http://invpy.com/chap20. If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/dodger.

```
 1. import pygame, random, sys
 2. from pygame.locals import *
 3.
 4. WINDOWWIDTH = 600
 5. WINDOWHEIGHT = 600
 6. TEXTCOLOR = (255, 255, 255)
 7. BACKGROUNDCOLOR = (0, 0, 0)
 8. FPS = 40
 9. BADDIEMINSIZE = 10
10. BADDIEMAXSIZE = 40
11. BADDIEMINSPEED = 1
12. BADDIEMAXSPEED = 8
13. ADDNEWBADDIERATE = 6
14. PLAYERMOVERATE = 5
15.
16. def terminate():
```

```
17.     pygame.quit()
18.     sys.exit()
19.
20. def waitForPlayerToPressKey():
21.     while True:
22.         for event in pygame.event.get():
23.             if event.type == QUIT:
24.                 terminate()
25.             if event.type == KEYDOWN:
26.                 if event.key == K_ESCAPE: # pressing escape quits
27.                     terminate()
28.                 return
29.
30. def playerHasHitBaddie(playerRect, baddies):
31.     for b in baddies:
32.         if playerRect.colliderect(b['rect']):
33.             return True
34.     return False
35.
36. def drawText(text, font, surface, x, y):
37.     textobj = font.render(text, 1, TEXTCOLOR)
38.     textrect = textobj.get_rect()
39.     textrect.topleft = (x, y)
40.     surface.blit(textobj, textrect)
41.
42. # set up pygame, the window, and the mouse cursor
43. pygame.init()
44. mainClock = pygame.time.Clock()
45. windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
46. pygame.display.set_caption('Dodger')
47. pygame.mouse.set_visible(False)
48.
49. # set up fonts
50. font = pygame.font.SysFont(None, 48)
51.
52. # set up sounds
53. gameOverSound = pygame.mixer.Sound('gameover.wav')
54. pygame.mixer.music.load('background.mid')
55.
56. # set up images
57. playerImage = pygame.image.load('player.png')
58. playerRect = playerImage.get_rect()
59. baddieImage = pygame.image.load('baddie.png')
60.
61. # show the "Start" screen
62. drawText('Dodger', font, windowSurface, (WINDOWWIDTH / 3), (WINDOWHEIGHT /
3))
```

```
63.  drawText('Press a key to start.', font, windowSurface, (WINDOWWIDTH / 3) -
30, (WINDOWHEIGHT / 3) + 50)
64.  pygame.display.update()
65.  waitForPlayerToPressKey()
66.
67.
68.  topScore = 0
69.  while True:
70.      # set up the start of the game
71.      baddies = []
72.      score = 0
73.      playerRect.topleft = (WINDOWWIDTH / 2, WINDOWHEIGHT - 50)
74.      moveLeft = moveRight = moveUp = moveDown = False
75.      reverseCheat = slowCheat = False
76.      baddieAddCounter = 0
77.      pygame.mixer.music.play(-1, 0.0)
78.
79.      while True: # the game loop runs while the game part is playing
80.          score += 1 # increase score
81.
82.          for event in pygame.event.get():
83.              if event.type == QUIT:
84.                  terminate()
85.
86.              if event.type == KEYDOWN:
87.                  if event.key == ord('z'):
88.                      reverseCheat = True
89.                  if event.key == ord('x'):
90.                      slowCheat = True
91.                  if event.key == K_LEFT or event.key == ord('a'):
92.                      moveRight = False
93.                      moveLeft = True
94.                  if event.key == K_RIGHT or event.key == ord('d'):
95.                      moveLeft = False
96.                      moveRight = True
97.                  if event.key == K_UP or event.key == ord('w'):
98.                      moveDown = False
99.                      moveUp = True
100.                 if event.key == K_DOWN or event.key == ord('s'):
101.                     moveUp = False
102.                     moveDown = True
103.
104.             if event.type == KEYUP:
105.                 if event.key == ord('z'):
106.                     reverseCheat = False
107.                     score = 0
108.                 if event.key == ord('x'):
```

```
109.                         slowCheat = False
110.                         score = 0
111.                    if event.key == K_ESCAPE:
112.                            terminate()
113.
114.                    if event.key == K_LEFT or event.key == ord('a'):
115.                        moveLeft = False
116.                    if event.key == K_RIGHT or event.key == ord('d'):
117.                        moveRight = False
118.                    if event.key == K_UP or event.key == ord('w'):
119.                        moveUp = False
120.                    if event.key == K_DOWN or event.key == ord('s'):
121.                        moveDown = False
122.
123.                if event.type == MOUSEMOTION:
124.                        # If the mouse moves, move the player where the cursor is.
125.                        playerRect.move_ip(event.pos[0] - playerRect.centerx,
event.pos[1] - playerRect.centery)
126.
127.            # Add new baddies at the top of the screen, if needed.
128.            if not reverseCheat and not slowCheat:
129.                baddieAddCounter += 1
130.            if baddieAddCounter == ADDNEWBADDIERATE:
131.                baddieAddCounter = 0
132.                baddieSize = random.randint(BADDIEMINSIZE, BADDIEMAXSIZE)
133.                newBaddie = {'rect': pygame.Rect(random.randint(0,
WINDOWWIDTH-baddieSize), 0 - baddieSize, baddieSize, baddieSize),
134.                             'speed': random.randint(BADDIEMINSPEED,
BADDIEMAXSPEED),
135.                             'surface':pygame.transform.scale(baddieImage,
(baddieSize, baddieSize)),
136.                             }
137.
138.                baddies.append(newBaddie)
139.
140.            # Move the player around.
141.            if moveLeft and playerRect.left > 0:
142.                playerRect.move_ip(-1 * PLAYERMOVERATE, 0)
143.            if moveRight and playerRect.right < WINDOWWIDTH:
144.                playerRect.move_ip(PLAYERMOVERATE, 0)
145.            if moveUp and playerRect.top > 0:
146.                playerRect.move_ip(0, -1 * PLAYERMOVERATE)
147.            if moveDown and playerRect.bottom < WINDOWHEIGHT:
148.                playerRect.move_ip(0, PLAYERMOVERATE)
149.
150.            # Move the mouse cursor to match the player.
151.            pygame.mouse.set_pos(playerRect.centerx, playerRect.centery)
```

```
152.
153.            # Move the baddies down.
154.            for b in baddies:
155.                if not reverseCheat and not slowCheat:
156.                    b['rect'].move_ip(0, b['speed'])
157.                elif reverseCheat:
158.                    b['rect'].move_ip(0, -5)
159.                elif slowCheat:
160.                    b['rect'].move_ip(0, 1)
161.
162.            # Delete baddies that have fallen past the bottom.
163.            for b in baddies[:]:
164.                if b['rect'].top > WINDOWHEIGHT:
165.                    baddies.remove(b)
166.
167.            # Draw the game world on the window.
168.            windowSurface.fill(BACKGROUNDCOLOR)
169.
170.            # Draw the score and top score.
171.            drawText('Score: %s' % (score), font, windowSurface, 10, 0)
172.            drawText('Top Score: %s' % (topScore), font, windowSurface, 10,
40)
173.
174.            # Draw the player's rectangle
175.            windowSurface.blit(playerImage, playerRect)
176.
177.            # Draw each baddie
178.            for b in baddies:
179.                windowSurface.blit(b['surface'], b['rect'])
180.
181.            pygame.display.update()
182.
183.            # Check if any of the baddies have hit the player.
184.            if playerHasHitBaddie(playerRect, baddies):
185.                if score > topScore:
186.                    topScore = score # set new top score
187.                break
188.
189.            mainClock.tick(FPS)
190.
191.    # Stop the game and show the "Game Over" screen.
192.    pygame.mixer.music.stop()
193.    gameOverSound.play()
194.
195.    drawText('GAME OVER', font, windowSurface, (WINDOWWIDTH / 3),
(WINDOWHEIGHT / 3))
```

```
196.        drawText('Press a key to play again.', font, windowSurface,
(WINDOWWIDTH / 3) - 80, (WINDOWHEIGHT / 3) + 50)
197.        pygame.display.update()
198.        waitForPlayerToPressKey()
199.
200.        gameOverSound.stop()
```

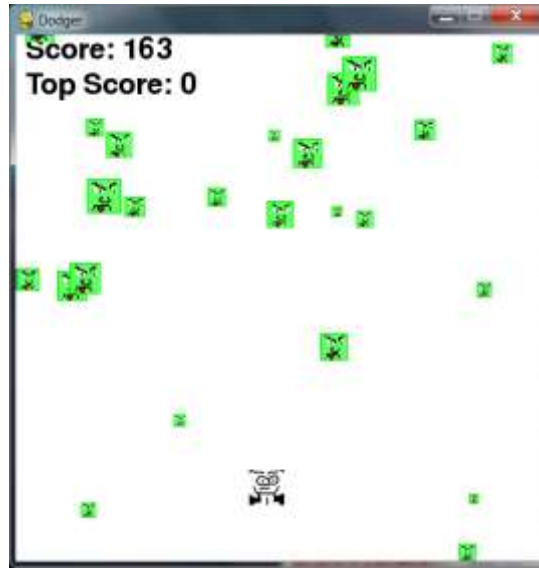When you run this program, the game will look like Figure 20-1.

Figure 20-1: An altered screenshot of the Dodger game.

Importing the Modules

```
1. import pygame, random, sys
2. from pygame.locals import *
```

The Dodger game imports the same modules previous Pygame programs have: `pygame`, `random`, `sys`, and `pygame.locals`. The `pygame.locals` module contains several constant variables that Pygame uses such as the event types (`QUIT`, `KEYDOWN`, etc.) and keyboard keys (`K_ESCAPE`, `K_LEFT`, etc.). By using the `from pygame.locals import *` syntax, you can just type `QUIT` in the source code instead of `pygame.locals.QUIT`.

Setting Up the Constant Variables

```
4. WINDOWWIDTH = 600
5. WINDOWHEIGHT = 600
```

```
 6. TEXTCOLOR = (255, 255, 255)
 7. BACKGROUNDCOLOR = (0, 0, 0)
```

The constant variables on lines 4 to 14 are much more descriptive than typing out the values. For example, from the line `windowSurface.fill(BACKGROUNDCOLOR)` is more understandable than `windowSurface.fill((0, 0, 0))`.

You can easily change the game by changing the constant variables. By changing `WINDOWWIDTH` on line 4, you automatically change the code everywhere `WINDOWWIDTH` is used. If you had used the value `600` instead, then you would have to change each occurrence of `600` in the code. It is easier to change the value in the constant once.

```
 8. FPS = 40
```

The `mainClock.tick()` method call on line 189 will slow the game down enough to be playable. You pass an integer to `mainClock.tick()` so that the function knows how long to pause the program. This integer (which you store in `FPS`) is the number of frames per second you want the game to run.

A "frame" is the drawing of graphics on the screen for a single iteration through the game loop. You can set `FPS` to 40, and always call `mainClock.tick(FPS)`. Then you can change `FPS` to a higher value to have the game run faster or a lower value to slow the game down.

```
 9. BADDIEMINSIZE = 10
10. BADDIEMAXSIZE = 40
11. BADDIEMINSPEED = 1
12. BADDIEMAXSPEED = 8
13. ADDNEWBADDIERATE = 6
```

Lines 9 to 13 set some more constant variables that will describe the falling baddies. The width and height of the baddies will be between `BADDIEMINSIZE` and `BADDIEMAXSIZE`. The rate at which the baddies fall down the screen will be between `BADDIEMINSPEED` and `BADDIEMAXSPEED` pixels per iteration through the game loop. And a new baddie will be added to the top of the window every `ADDNEWBADDIERATE` iterations through the game loop.

```
14. PLAYERMOVERATE = 5
```

The `PLAYERMOVERATE` will store the number of pixels the player's character moves in the window on each iteration through the game loop if the character is moving. By increasing this number, you can increase the speed the character moves.

Defining Functions

There are several functions you'll create for the game:

```
16. def terminate():
17.     pygame.quit()
18.     sys.exit()
```

Pygame requires that you call both `pygame.quit()` and `sys.exit()`. Put them both into a function called `terminate()`. Now you only need to call `terminate()`, instead of both of the `pygame.quit()` and `sys.exit()` functions.

```
20. def waitForPlayerToPressKey():
21.     while True:
22.         for event in pygame.event.get():
```

Sometimes you'll want to pause the game until the player presses a key. Create a new function called `waitForPlayerToPressKey()`. Inside this function, there's an infinite loop that only breaks when a `KEYDOWN` or `QUIT` event is received. At the start of the loop, `pygame.event.get()` to return a list of Event objects to check out.

```
23.             if event.type == QUIT:
24.                 terminate()
```

If the player has closed the window while the program is waiting for the player to press a key, Pygame will generate a `QUIT` event. In that case, call the `terminate()` function on line 24.

```
25.             if event.type == KEYDOWN:
26.                 if event.key == K_ESCAPE: # pressing escape quits
27.                     terminate()
28.                 return
```

If you receive a `KEYDOWN` event, then you should first check if it is the ESC key that was pressed. If the player presses the ESC key, the program should terminate. If that wasn't the case, then execution will skip the if-block on line 27 and go straight to the `return` statement, which exits the `waitForPlayerToPressKey()` function.

If a `QUIT` or `KEYDOWN` event isn't generated, then the code keeps looping. Since the loop does nothing, this will make it look like the game has frozen until the player presses a key.

```
30. def playerHasHitBaddie(playerRect, baddies):
31.     for b in baddies:
```

```
32.             if playerRect.colliderect(b['rect']):
33.                 return True
34.     return False
```

The `playerHasHitBaddie()` function will return `True` if the player's character has collided with one of the baddies. The `baddies` parameter is a list of "baddie" dictionary data structures. Each of these dictionaries has a `'rect'` key, and the value for that key is a Rect object that represents the baddie's size and location.

`playerRect` is also a Rect object. Rect objects have a method named `colliderect()` that returns `True` if the Rect object has collided with the Rect object that is passed to it. Otherwise, `colliderect()` will return `False`.

The `for` loop on line 31 iterates through each baddie dictionary in the `baddies` list. If any of these baddies collide with the player's character, then `playerHasHitBaddie()` will return `True`. If the code manages to iterate through all the baddies in the `baddies` list without detecting a collision with any of them, it will return `False`.

```
36. def drawText(text, font, surface, x, y):
37.     textobj = font.render(text, 1, TEXTCOLOR)
38.     textrect = textobj.get_rect()
39.     textrect.topleft = (x, y)
40.     surface.blit(textobj, textrect)
```

Drawing text on the window involves a few steps. First, the `render()` method call on line 37 creates a Surface object that has the text rendered in a specific font on it.

Next, you need to know the size and location of the Surface object. You can get a Rect object with this information from the `get_rect()` Surface method.

The Rect object returned on line 38 from `get_rect()` has a copy of the width and height information from the Surface object. Line 39 changes the location of the Rect object by setting a new tuple value for its `topleft` attribute.

Finally, line 40 blits the Surface object of the rendered text onto the Surface object that was passed to the `drawText()` function. Displaying text in Pygame take a few more steps than simply calling the `print()` function. But if you put this code into a single function named `drawText()`, then you only need to call this function to display text on the screen.

Initializing Pygame and Setting Up the Window

Now that the constant variables and functions are finished, start calling the Pygame functions that set up the window and clock.

```
42. # set up pygame, the window, and the mouse cursor
43. pygame.init()
44. mainClock = pygame.time.Clock()
```

Line 43 sets up the Pygame by calling the pygame.init() function. Line 44 creates a pygame.time.Clock() object and stores it in the mainClock variable. This object will help us keep the program from running too fast.

```
45. windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
```

Line 45 creates a new Surface object which is used for the window displayed on the screen. You can specify the width and height of this Surface object (and the window) by passing a tuple with the WINDOWWIDTH and WINDOWHEIGHT constant variables. Notice that there's only one argument passed to pygame.display.set_mode(): a tuple. The arguments for pygame.display.set_mode() are not two integers but one tuple of two integers.

```
46. pygame.display.set_caption('Dodger')
```

Line 46 sets the caption of the window to the string 'Dodger'. This caption will appear in the title bar at the top of the window.

```
47. pygame.mouse.set_visible(False)
```

In Dodger, the mouse cursor shouldn't be visible. This is because you want the mouse to be able to move the player's character around the screen, but the mouse cursor would get in the way of the character's image on the screen. Calling pygame.mouse.set_visible(False) will tell Pygame to make the cursor not visible.

Fullscreen Mode

The pygame.display.set_mode() function has a second, optional parameter. You can pass the pygame.FULLSCREEN constant to make the window take up the entire screen instead of being in a window. Look at this modification to line 45:

```
45. windowSurface = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT),
pygame.FULLSCREEN)
```

It will still be WINDOWWIDTH and WINDOWHEIGHT in size for the windows width and height, but the image will be stretched larger to fit the screen. Try running the program wiuth and without fullscreen mode.

```
49. # set up fonts
50. font = pygame.font.SysFont(None, 48)
```

Line 49 creates a Font object to use by calling `pygame.font.SysFont()`. Passing `None` uses the default font. Passing `48` makes the font have a size of 48 points.

```
52. # set up sounds
53. gameOverSound = pygame.mixer.Sound('gameover.wav')
54. pygame.mixer.music.load('background.mid')
```

Next, create the Sound objects and set up the background music. The background music will constantly be playing during the game, but Sound objects will only be played when the player loses the game.

You can use any *.wav* or *.mid* file for this game. Some sound files are available at this book's website at http://invpy.com/downloads. Or you can use your own sound files for this game, as long as they have the filenames of *gameover.wav* and *background.mid*. (You can change the strings used on lines 53 and 54 to match the filenames.)

The `pygame.mixer.Sound()` constructor function creates a new Sound object and stores a reference to this object in the `gameOverSound` variable. In your own games, you can create as many Sound objects as you like, each with a different sound file.

The `pygame.mixer.music.load()` function loads a sound file to play for the background music. This function doesn't return any objects, and only one background sound file can be loaded at a time.

```
56. # set up images
57. playerImage = pygame.image.load('player.png')
58. playerRect = playerImage.get_rect()
59. baddieImage = pygame.image.load('baddie.png')
```

Next you'll load the image files to be used for the player's character and the baddies on the screen. The image for the character is stored in *player.png* and the image for the baddies is stored in *baddie.png*. All the baddies look the same, so you only need one image file for them. You can download these images from this book's website at http://invpy.com/downloads.

Display the Start Screen

When the game first starts, display the "Dodger" name on the screen. You also want to instruct the player that they can start the game by pushing any key. This screen appears so that the player has time to get ready to start playing after running the program.

```
61. # show the "Start" screen
62. drawText('Dodger', font, windowSurface, (WINDOWWIDTH / 3), (WINDOWHEIGHT /
3))
63. drawText('Press a key to start.', font, windowSurface, (WINDOWWIDTH / 3) -
30, (WINDOWHEIGHT / 3) + 50)
64. pygame.display.update()
65. waitForPlayerToPressKey()
```

On lines 62 and 63, call the drawText() function and pass it five arguments:

1) The string of the text you want to appear.

2) The font that you want the string to appear in.

3) The Surface object onto which to render the text.

4) The X coordinate on the Surface object to draw the text at.

5) The Y coordinate on the Surface object to draw the text at.

This may seem like many arguments to pass for a function call, but keep in mind that this function call replaces five lines of code each time you call it. This shortens the program and makes it easier to find bugs since there's less code to check.

The waitForPlayerToPressKey() function will pause the game by looping until a KEYDOWN event is generated. Then the execution breaks out of the loop and the program continues to run.

Start of the Main Game Code

```
68. topScore = 0
69. while True:
```

The value in the topScore variable starts at 0 when the program first runs. Whenever the player loses and has a score larger than the current top score, the top score is replaced with this larger score.

The infinite loop started on line 69 is technically not the "game loop". The game loop handles events and drawing the window while the game is running. Instead, this while loop will iterate each time the player starts a new game. When the player loses and the game resets, the program's execution will loop back to line 69.

```
70.     # set up the start of the game
71.     baddies = []
72.     score = 0
```

At the beginning, you want to set `baddies` to an empty list. The `baddies` variable is a list of dictionary objects with the following keys:

- `'rect'` - The Rect object that describes where and what size the baddie is.
- `'speed'` - How fast the baddie falls down the screen. This integer represents pixels per iteration through the game loop.
- `'surface'` - The Surface object that has the scaled baddie image drawn on it. This is the Surface object that is blitted to the Surface object returned by `pygame.display.set_mode()`.

Line 72 resets the player's score to `0`.

```
73.        playerRect.topleft = (WINDOWWIDTH / 2, WINDOWHEIGHT - 50)
```

The starting location of the player is in the center of the screen and 50 pixels up from the bottom. The first item in line 73's tuple is the X-coordinate of the left edge. The second item in the tuple is the Y-coordinate of the top edge.

```
74.        moveLeft = moveRight = moveUp = moveDown = False
75.        reverseCheat = slowCheat = False
76.        baddieAddCounter = 0
```

The movement variables `moveLeft`, `moveRight`, `moveUp`, and `moveDown` are set to `False`. The `reverseCheat` and `slowCheat` variables are also set to `False`. They will be set to `True` only when the player enables these cheats by holding down the "z" and "x" keys, respectively.

The `baddieAddCounter` variable is a counter to tell the program when to add a new baddie at the top of the screen. The value in `baddieAddCounter` increments by one each time the game loop iterates.

When `baddieAddCounter` is equal to `ADDNEWBADDIERATE`, then the `baddieAddCounter` counter resets to 0 and a new baddie is added to the top of the screen. (This check is done later on line 130.)

```
77.        pygame.mixer.music.play(-1, 0.0)
```

The background music starts playing on line 77 with a call to `pygame.mixer.music.play()`. The first argument is the number of times the music should repeat itself. `-1` is a special value that tells Pygame you want the music to repeat endlessly.

The second argument is a float that says how many seconds into the music you want it to start playing. Passing `0.0` means the music starts playing from the beginning.

The Game Loop

The game loop's code constantly updates the state of the game world by changing the position of the player and baddies, handling events generated by Pygame, and drawing the game world on the screen. All of this happens several dozen times a second, which makes it run in "real time".

```
79.     while True: # the game loop runs while the game part is playing
80.         score += 1 # increase score
```

Line 79 is the start of the main game loop. Line 80 increases the player's score on each iteration of the game loop. The longer the player can go without losing, the higher their score. The loop will only exit when the player either loses the game or quits the program.

Event Handling

There are four different types of events the program will handle: QUIT, KEYDOWN, KEYUP, and MOUSEMOTION.

```
82.         for event in pygame.event.get():
83.             if event.type == QUIT:
84.                 terminate()
```

Line 82 is the start of the event-handling code. It calls pygame.event.get(), which returns a list of Event objects. Each Event object represents an event that has happened since the last call to pygame.event.get(). The code will check the type attribute of the event object to see what type of event it is, and handle the event accordingly.

If the type attribute of the Event object is equal to QUIT, then the user has closed the program. The QUIT constant variable was imported from the pygame.locals module.

```
86.             if event.type == KEYDOWN:
87.                 if event.key == ord('z'):
88.                     reverseCheat = True
89.                 if event.key == ord('x'):
90.                     slowCheat = True
```

If the event's type is KEYDOWN, the player has pressed down a key. The Event object for keyboard events will also have a key attribute that is set to the integer ordinal value of the key pressed. The ord() function will return the ordinal value of the letter passed to it.

For example, line 87 checks if the event describes the "z" key being pressed down with event.key == ord('z'). If this condition is True, set the reverseCheat variable to True to

indicate that the reverse cheat has been activated. Line 89 checks if the "x" key has been pressed to activate the slow cheat.

Pygame's keyboard events always use the ordinal values of lowercase letters, not uppercase. Always use `event.key == ord('z')` instead of `event.key == ord('Z')`. Otherwise, your program may act as though the key wasn't pressed.

```
 91.                    if event.key == K_LEFT or event.key == ord('a'):
 92.                        moveRight = False
 93.                        moveLeft = True
 94.                    if event.key == K_RIGHT or event.key == ord('d'):
 95.                        moveLeft = False
 96.                        moveRight = True
 97.                    if event.key == K_UP or event.key == ord('w'):
 98.                        moveDown = False
 99.                        moveUp = True
100.                    if event.key == K_DOWN or event.key == ord('s'):
101.                        moveUp = False
102.                        moveDown = True
```

Lines 91 to 102 check if the event was generated by the player pressing one of the arrow or WASD keys. There isn't an ordinal value for every key on the keyboard, such as the arrow keys or the ESC key. Instead, the `pygame.locals` module provides constant variables to use instead.

Line 91 checks if the player has pressed the left arrow key with `event.key == K_LEFT`. Notice that pressing down on one of the arrow keys not only sets a movement variable to `True`, but it also sets the movement variable in the opposite direction to `False`.

For example, if the left arrow key is pushed down, then the code on line 93 sets `moveLeft` to `True`, but it also sets `moveRight` to `False`. This prevents the player from confusing the program into thinking that the player's character should move in two opposite directions at the same time.

Table 20-1 lists commonly-used constant variables for the `key` attribute of keyboard-related Event objects.

Table 20-1: Constant Variables for Keyboard Keys

Pygame Constant Variable	Keyboard Key	Pygame Constant Variable	Keyboard Key
K_LEFT	Left arrow	K_HOME	Home
K_RIGHT	Right arrow	K_END	End
K_UP	Up arrow	K_PAGEUP	PgUp

K_DOWN	Down arrow	K_PAGEDOWN	PgDn
K_ESCAPE	Esc	K_F1	F1
K_BACKSPACE	Backspace	K_F2	F2
K_TAB	Tab	K_F3	F3
K_RETURN	Return or Enter	K_F4	F4
K_SPACE	Space bar	K_F5	F5
K_DELETE	Del	K_F6	F6
K_LSHIFT	Left Shift	K_F7	F7
K_RSHIFT	Right Shift	K_F8	F8
K_LCTRL	Left Ctrl	K_F9	F9
K_RCTRL	Right Ctrl	K_F10	F10
K_LALT	Left Alt	K_F11	F11
K_RALT	Right Alt	K_F12	F12

```
104.             if event.type == KEYUP:
105.                 if event.key == ord('z'):
106.                     reverseCheat = False
107.                     score = 0
108.                 if event.key == ord('x'):
109.                     slowCheat = False
110.                     score = 0
```

The KEYUP event is created whenever the player stops pressing down on a keyboard key and releases it. Event objects with a type of KEYUP also have a key attribute just like KEYDOWN events.

Line 105 checks if the player has released the "z" key, which will deactivate the reverse cheat. In that case, line 106 sets reverseCheat to False and line 107 resets the score to 0. The score reset is to discourage the player for using the cheats.

Lines 108 to 110 do the same thing for the "x" key and the slow cheat. When the "x" key is released, slowCheat is set to False and the player's score is reset to 0.

```
111.                    if event.key == K_ESCAPE:
112.                        terminate()
```

At any time during the game, the player can press the ESC key on the keyboard to quit. Line 14 checks if the key that was released was the ESC key by checking `event.key == K_ESCAPE`. If so, line 112 calls the `terminate()` function to exit the program.

```
114.                    if event.key == K_LEFT or event.key == ord('a'):
115.                        moveLeft = False
116.                    if event.key == K_RIGHT or event.key == ord('d'):
117.                        moveRight = False
118.                    if event.key == K_UP or event.key == ord('w'):
119.                        moveUp = False
120.                    if event.key == K_DOWN or event.key == ord('s'):
121.                        moveDown = False
```

Lines 114 to 121 check if the player has stopped holding down one of the arrow or WASD keys. In that case, the code sets the corresponding movement variable to `False`.

For example, if the player was holding down the left arrow key, then the `moveLeft` would have been set to `True` on line 93. When they release it, the condition on line 114 will evaluate to `True`, and the `moveLeft` variable will be set to `False`.

The `move_ip()` Method

```
123.               if event.type == MOUSEMOTION:
124.                   # If the mouse moves, move the player where the cursor is.
125.                   playerRect.move_ip(event.pos[0] - playerRect.centerx,
event.pos[1] - playerRect.centery)
```

Now that you've handled the keyboard events, let's handle any mouse events that may have been generated. The Dodger game doesn't do anything if the player has clicked a mouse button, but it does respond when the player moves the mouse. This gives the player two ways of controlling the player character in the game: the keyboard or the mouse.

The MOUSEMOTION event is generated whenever the mouse is moved. Event objects with a `type` set to MOUSEMOTION also have an attribute named `pos` for the position of the mouse event. The `pos` attribute stores a tuple of the X- and Y-coordinates of where the mouse cursor moved in the window. If the event's type is MOUSEMOTION, the player's character moves to the position of the mouse cursor.

The `move_ip()` method for Rect objects will move the location of the Rect object horizontally or vertically by a number of pixels. For example, `playerRect.move_ip(10, 20)` would move the Rect object 10 pixels to the right and 20 pixels down. To move the Rect object left or up, pass negative values. For example, `playerRect.move_ip(-5, -15)` will move the Rect object left by 5 pixels and up 15 pixels.

The "ip" at the end of `move_ip()` stands for "in place". This is because the method changes the Rect object itself, rather than return a new Rect object with the changes. There is also a `move()` method which doesn't change the Rect object, but instead creates and returns a new Rect object in the new location.

Adding New Baddies

```
127.           # Add new baddies at the top of the screen, if needed.
128.          if not reverseCheat and not slowCheat:
129.              baddieAddCounter += 1
```

On each iteration of the game loop, increment the `baddieAddCounter` variable by one. This only happens if the cheats are not enabled. Remember that `reverseCheat` and `slowCheat` are set to `True` as long as the "z" and "x" keys are being held down, respectively

And while those keys are being held down, `baddieAddCounter` isn't incremented. Therefore, no new baddies will appear at the top of the screen.

```
130.          if baddieAddCounter == ADDNEWBADDIERATE:
131.              baddieAddCounter = 0
132.              baddieSize = random.randint(BADDIEMINSIZE, BADDIEMAXSIZE)
133.              newBaddie = {'rect': pygame.Rect(random.randint(0,
WINDOWWIDTH-baddieSize), 0 - baddieSize, baddieSize, baddieSize),
134.                          'speed': random.randint(BADDIEMINSPEED,
BADDIEMAXSPEED),
135.                          'surface':pygame.transform.scale(baddieImage,
(baddieSize, baddieSize)),
136.                          }
```

When the `baddieAddCounter` reaches the value in `ADDNEWBADDIERATE`, it is time to add a new baddie to the top of the screen. First, the `baddieAddCounter` counter is reset back to `0`.

Line 132 generates a size for the baddie in pixels. The size will be a random integer between `BADDIEMINSIZE` and `BADDIEMAXSIZE`, which are constants set to `10` and `40` on lines 9 and 10.

Line 133 is where a new baddie data structure is created. Remember, the data structure for `baddies` is simply a dictionary with keys `'rect'`, `'speed'`, and `'surface'`. The `'rect'` key

holds a reference to a Rect object which stores the location and size of the baddie. The call to the `pygame.Rect()` constructor function has four parameters: the X-coordinate of the top edge of the area, the Y-coordinate of the left edge of the area, the width in pixels, and the height in pixels.

The baddie needs to appear randomly across the top of the window, so pass `random.randint(0, WINDOWWIDTH-baddieSize)` for the X-coordinate of the left edge. The reason you pass `WINDOWWIDTH-baddieSize` instead of `WINDOWWIDTH` is because this value is for the left edge of the baddie. If the left edge of the baddie is too far on the right side of the screen, then part of the baddie will be off the edge of the window and not visible.

The bottom edge of the baddie should be just above the top edge of the window. The Y-coordinate of the top edge of the window is 0. To put the baddie's bottom edge there, set the top edge to `0 - baddieSize`.

The baddie's width and height should be the same (the image is a square), so pass `baddieSize` for the third and fourth argument.

The rate of speed that the baddie moves down the screen is set in the `'speed'` key. Set it to a random integer between `BADDIEMINSPEED` and `BADDIEMAXSPEED`.

```
138.                baddies.append(newBaddie)
```

Line 138 will add the newly created baddie data structure to the list of baddie data structures. The program will use this list to check if the player has collided with any of the baddies, and to know where to draw baddies on the window.

Moving the Player's Character

```
140.          # Move the player around.
141.          if moveLeft and playerRect.left > 0:
142.              playerRect.move_ip(-1 * PLAYERMOVERATE, 0)
```

The four movement variables `moveLeft`, `moveRight`, `moveUp` and `moveDown` are set to `True` and `False` when Pygame generates the `KEYDOWN` and `KEYUP` events, respectively.

If the player's character is moving left and the left edge of the player's character is greater than `0` (which is the left edge of the window), then `playerRect` should be moved to the left.

You'll always move the `playerRect` object by the number of pixels in `PLAYERMOVERATE`. To get the negative form of an integer, multiple it by `-1`. On line 142, since `5` is stored in `PLAYERMOVERATE`, the expression `-1 * PLAYERMOVERATE` evaluates to `-5`.

Therefore, calling playerRect.move_ip(-1 * PLAYERMOVERATE, 0) will change the location of playerRect by 5 pixels to the left of its current location.

```
143.            if moveRight and playerRect.right < WINDOWWIDTH:
144.                playerRect.move_ip(PLAYERMOVERATE, 0)
145.            if moveUp and playerRect.top > 0:
146.                playerRect.move_ip(0, -1 * PLAYERMOVERATE)
147.            if moveDown and playerRect.bottom < WINDOWHEIGHT:
148.                playerRect.move_ip(0, PLAYERMOVERATE)
```

Lines 143 to 148 do the same thing for the other three directions: right, up, and down. Each of the three if statements in lines 143 to 148 checks that their movement variable is set to True and that the edge of the Rect object of the player is inside the window. Then it calls move_ip() to move the Rect object.

The pygame.mouse.set_pos() Function

```
150.            # Move the mouse cursor to match the player.
151.            pygame.mouse.set_pos(playerRect.centerx, playerRect.centery)
```

Line 151 moves the mouse cursor to the same position as the player's character. The pygame.mouse.set_pos() function moves the mouse cursor to the X- and Y-coordinates you pass it. This is so that the mouse cursor and player's character are always in the same place.

Specifically, the cursor will be right in the middle of the character's Rect object because you passed the centerx and centery attributes of playerRect for the coordinates. The mouse cursor still exists and can be moved, even though it is invisible because of the pygame.mouse.set_visible(False) call on line 47.

```
153.            # Move the baddies down.
154.            for b in baddies:
```

Now loop through each baddie data structure in the baddies list to move them down a little.

```
155.                if not reverseCheat and not slowCheat:
156.                    b['rect'].move_ip(0, b['speed'])
```

If neither of the cheats have been activated, then move the baddie's location down a number of pixels equal to its speed, which is stored in the 'speed' key.

Implementing the Cheat Codes

```
157.                 elif reverseCheat:
158.                     b['rect'].move_ip(0, -5)
```

If the reverse cheat is activated, then the baddie should move up by five pixels. Passing -5 for the second argument to move_ip() will move the Rect object upwards by five pixels.

```
159.                 elif slowCheat:
160.                     b['rect'].move_ip(0, 1)
```

If the slow cheat has been activated, then the baddie should move downwards, but only by the slow speed of one pixel per iteration through the game loop. The baddie's normal speed (which is stored in the 'speed' key of the baddie's data structure) is ignored while the slow cheat is activated.

Removing the Baddies

```
162.         # Delete baddies that have fallen past the bottom.
163.         for b in baddies[:]:
```

Any baddies that fell below the bottom edge of the window should be removed from the baddies list. Remember that while iterating through a list, do not modify the contents of the list by adding or removing items. So instead of iterating through the baddies list with the for loop, iterate through a copy of the baddies list. This copy is made with the blank slicing operator [:].

The for loop on line 163 uses a variable b for the current item in the iteration through baddies[:].

```
164.             if b['rect'].top > WINDOWHEIGHT:
165.                 baddies.remove(b)
```

Let's evaluate the expression b['rect'].top. b is the current baddie data structure from the baddies[:] list. Each baddie data structure in the list is a dictionary with a 'rect' key, which stores a Rect object. So b['rect'] is the Rect object for the baddie.

Finally, the top attribute is the Y-coordinate of the top edge of the rectangular area. Remember that the Y-coordinates increase going down. So b['rect'].top > WINDOWHEIGHT will check if the top edge of the baddie is below the bottom of the window.

If this condition is True, then line 165 removes the baddie data structure from the baddies list.

Drawing the Window

After all the data structures have been updated, the game world should be drawn using Pygame's image functions. Because the game loop is executed several times a second, drawing the baddies and player in new positions makes their movement look smooth and natural.

```
167.          # Draw the game world on the window.
168.          windowSurface.fill(BACKGROUNDCOLOR)
```

First, before drawing anything else, line 168 blacks out the entire screen to erase anything drawn on it previously.

Remember that the Surface object in `windowSurface` is the special Surface object because it was the one returned by `pygame.display.set_mode()`. Therefore, anything drawn on that Surface object will appear on the screen after `pygame.display.update()` is called.

Drawing the Player's Score

```
170.          # Draw the score and top score.
171.          drawText('Score: %s' % (score), font, windowSurface, 10, 0)
172.          drawText('Top Score: %s' % (topScore), font, windowSurface, 10,
40)
```

Lines 171 and 172 render the text for the score and top score to the top left corner of the window. The `'Score: %s' % (score)` expression uses string interpolation to insert the value in the `score` variable into the string.

Pass this string, the Font object stored in the `font` variable, the Surface object on which to draw the text on, and the X- and Y-coordinates of where the text should be placed. The `drawText()` will handle the call to the `render()` and `blit()` methods.

For the top score, do the same thing. Pass 40 for the Y-coordinate instead of 0 so that the top score text appears beneath the score text.

Drawing the Player's Character

```
174.          # Draw the player's rectangle
175.          windowSurface.blit(playerImage, playerRect)
```

The information about the player is kept in two different variables. `playerImage` is a Surface object that contains all the colored pixels that make up the player's character's image.

`playerRect` is a Rect object that stores the information about the size and location of the player's character.

The `blit()` method draws the player character's image (in `playerImage`) on `windowSurface` at the location in `playerRect`.

```
177.          # Draw each baddie
178.          for b in baddies:
179.              windowSurface.blit(b['surface'], b['rect'])
```

Line 178's `for` loop draws every baddie on the `windowSurface` object. Each item in the `baddies` list is a dictionary. The dictionaries' `'surface'` and `'rect'` keys contain the Surface object with the baddie image and the Rect object with the position and size information, respectively.

```
181.          pygame.display.update()
```

Now that everything has been drawn to `windowSurface`, draw this Surface object to the screen by calling `pygame.display.update()`.

Collision Detection

```
183.          # Check if any of the baddies have hit the player.
184.          if playerHasHitBaddie(playerRect, baddies):
185.              if score > topScore:
186.                  topScore = score # set new top score
187.              break
```

Lines 184 checks if the player has collided with any baddies by calling `playerHasHitBaddie()`. This function will return `True` if the player's character has collided with any of the baddies in the `baddies` list. Otherwise, the function will return `False`.

If the player's character has hit a baddie, lines 185 and 186 update the top score if the current score is greater than it. Then the execution breaks out of the game loop at line 187. The program's execution will move to line 191.

```
189.          mainClock.tick(FPS)
```

To keep the computer from running through the game loop as fast as possible (which would be much too fast for the player to keep up with), call `mainClock.tick()` to pause for a brief amount of time. The pause will be long enough to ensure that about 40 (the value stored inside the `FPS` variable) iterations through the game loop occur each second.

The Game Over Screen

```
191.        # Stop the game and show the "Game Over" screen.
192.        pygame.mixer.music.stop()
193.        gameOverSound.play()
```

When the player loses, the game stops playing the background music and plays the "game over" sound effect. Line 192 calls the `stop()` function in the `pygame.mixer.music` module to stop the background music. Line 193 calls the `play()` method on the Sound object stored in `gameOverSound`.

```
195.        drawText('GAME OVER', font, windowSurface, (WINDOWWIDTH / 3),
(WINDOWHEIGHT / 3))
196.        drawText('Press a key to play again.', font, windowSurface,
(WINDOWWIDTH / 3) - 80, (WINDOWHEIGHT / 3) + 50)
197.        pygame.display.update()
198.        waitForPlayerToPressKey()
```

Lines 195 and 196 call the `drawText()` function to draw the "game over" text to the `windowSurface` object. Line 197 calls `pygame.display.update()` to draw this Surface object to the screen. After displaying this text, the game stops until the player presses a key by calling the `waitForPlayerToPressKey()` function.

```
200.        gameOverSound.stop()
```

After the player presses a key, the program execution will return from the `waitForPlayerToPressKey()` call on line 198. Depending on how long the player takes to press a key, the "game over" sound effect may or may not still be playing. To stop this sound effect before a new game starts, line 200 calls `gameOverSound.stop()`.

Modifying the Dodger Game

That's it for our graphical game. You may find that the game is too easy or too hard. But the game is easy to modify because we took the time to use constant variables instead of typing in the values directly. Now all we need to do to change the game is modify the value set in the constant variables.

For example, if you want the game to run slower in general, change the `FPS` variable on line 8 to a smaller value such as 20. This will make both the baddies and the player's character move slower since the game loop will only be executed 20 times a second instead of 40.

If you just want to slow down the baddies and not the player, then change BADDIEMAXSPEED to a smaller value such as 4. This will make all the baddies move between 1 (the value in BADDIEMINSPEED) and 4 pixels per iteration through the game loop instead of 1 and 8.

If you want the game to have fewer but larger baddies instead of many fast baddies, then increase ADDNEWBADDIERATE to 12, BADDIEMINSIZE to 40, and BADDIEMAXSIZE to 80. Now that baddies are being added every 12 iterations through the game loop instead of every 6 iterations, there will be half as many baddies as before. But to keep the game interesting, the baddies are now much larger than before.

While the basic game remains the same, you can modify any of the constant variables to drastically affect the behavior of the game. Keep trying out new values for the constant variables until you find a set of values you like the best.

Summary

Unlike our previous text-based games, Dodger really looks like the kind of modern computer game we usually play. It has graphics and music and uses the mouse. While Pygame provides functions and data types as building blocks, it is you the programmer who puts them together to create fun, interactive games.

And it is all because you know how to instruct the computer to do it, step by step, line by line. You can speak the computer's language, and get it to do large amounts of number crunching and drawing for you. This is a useful skill, and I hope you'll continue to learn more about Python programming. (And there's still much more to learn!)

Here are several websites that can teach you more about programming Python:

- http://reddit.com/r/inventwithpython – This site has several users who could help you with the material in this book.
- http://inventwithpython.com - This book's website, which includes all the source code for these programs and additional information. This site also has the image and sound files used in the Pygame programs.
- http://inventwithpython.com/pygame – My second book, *Making Games with Python & Pygame*, which covers Pygame in more detail. It's free to download and has the source code for many more games.
- http://inventwithpython.com/hacking – My third book, *Hacking Secret Ciphers with Python*, which covers more cryptography and code breaking programs. It's also free to download.
- http://inventwithpython.com/automate – My fourth book, *Automate the Boring Stuff with Python*, which teaches you practical programming skills. It's also free to download.

- http://python.org/doc/ - More Python tutorials and the documentation of all the Python modules and functions.
- http://pygame.org/docs/ - Complete documentation on the modules and functions for Pygame.
- al@inventwithpython.com - My email address. Feel free to email me your questions about this book or about Python programming.

Or you can find out more about Python by searching the web. Go to http://google.com and search for "Python programming" or "Python tutorials" to find websites that can teach you more about Python programming.

Now get going and invent your own games. And good luck!