

Invent Your Own Computer Games with Python

2nd Edition

Al Sweigart

*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 this book comes from a gap I saw in today's literature for kids interested in learning to program. I started programming when I was 9 years old in the BASIC 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 that is used by programmers professionally.

The current crop of programming books for kids that I've seen fell into two categories. First, books that did not teach programming so much as "game creation software" or a dumbed-down language 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 have 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. Thank you again for reading this book, and feel free to email me any questions or comments.

Al Sweigart
al@inv8mptsm2(dm)-2(m)8oapaT83/R8 ll 0 0

Table of Contents

Source Code Listing

hello.py	21
guess.py	30
jokes.py	51
dragon.py	58
buggy.py	83
coinFlips.py	87
hangman.py	103
tictactoe.py	150
truefalsefizz.py	172
bagels.py	184
sonar.py	213
cipher.py	244
reversi.py	261
aisim1.py	292
aisim2.py	294
aisim3.py	299
pygameHelloWorld.py	309
animation.py	324
collisionDetection.py	338
pygameInput.py	348
spritesAndSounds.py	360
dodger.py	371

1	Installing Python	1
	Downloading and Installing Python	2
	Starting Python	4
	How to Use This Book	4
	The Featured Programs	5
	Line Numbers and Spaces	5
	Summary	7
2	The Interactive Shell	8
	Some Simple Math Stuff	8
	Evaluating Expressions	11
	Storing Values in Variables	12

	Quotes and Double Quotes	53
	The end Keyword Argument	54
	Summary	55
6	Dragon Realm	56
	Introducing Functions	56
	Sample Run of Dragon Realm	57
	Dragon Realm's Source Code	57
	def Statements	60
	Boolean Operators	61
	Return Values	65
	Variable Scope	65
	Parameters	68
	Where to Put Function Definitions	70
	Displaying the Game Results	71
	The Colon :	73
	Where the Program Really Begins	73
	Designing the Program	75
	Summary	76
7	Using the Debugger	77
	Bugs!	77
	Starting the Debugger	78
	Stepping	80
	The Go and Quit Buttons	81
	Stepping Over and Stepping Out	81
	Find the Bug	83
	Break Points	86
	Summary	88
8	Flow Charts	89
	How to Play "Hangman"	89
	Sample Run of "Hangman"	89
	ASCII Art	91
	Designing a Program with a Flowchart	92
	Creating the Flow Chart	93

Summary: The Importance of Planning Out the Game

	Summary: Creating Game-Playing Artificial Intelligences	182
11	Bagels	183
	Sample Run	184
	Bagel's Source Code	184
	Designing the Program	186
	The <code>random.shuffle()</code> Function	188
	Augmented Assignment Operators	190
	The <code>sort()</code> List Method	192
	The <code>join()</code> String Method	192
	String Interpolation	194
	Summary: Getting Good at Bagels	198
12	Cartesian Coordinates	200
	Grids and Cartesian Coordinates	201
	Negative Numbers	202
	202	

Brute Force	251
Summary: Reviewing Our Caesar Cipher Program	253
15 Reversi	256
How to Play Reversi	255
Sample Run	257
Reversi's Source Code	260
The <code>bool()</code> Function	276
Summary: Reviewing the Reversi Game	290
16 AI Simulation	291
"Computer vs. Computer" Games	291
AI <code>Sim1.py</code> Source Code	292
AI <code>Sim2.py</code> Source Code	294
Percentages	296
The <code>round()</code> Function	297
Comparing Different AI Algorithms	299
AI <code>Sim3.py</code> Source Code	299
Learning New Things by Running Simulation Experiments	305
17 Graphics and Animation	306
Installing Pygame	307
Hello World in Pygame	308
Hello World's Source Code	308
Importing the Pygame Module	311
Variables Store References to Objects	313
Colors in Pygame	313
Fonts, and the <code>pygame.font.SysFont()</code> Function	315
Attributes	316
Constructor Functions and the <code>type()</code> function.	317
The <code>pygame.PixelArray</code> Data Type	321
Events and the Game Loop	322
Animation	324
The Animation Program's Source Code	324
Some Small Modifications	335
Summary: Pygame Programming	335



Topics Covered In This Chapter:

- Downloading and installing the Python interpreter.
- Using IDLE's interactive shell to run instructions.
- How to use this book.
- The book's website at <http://inventwithpython.com>

Hello! This is a book that will teach you how to program by showing you how to create computer 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 software you'll need is free and you can download it from the Internet.

When I was a kid, I found a book like this that taught me how to write my first programs and games. It was fun and easy. Now as an adult, I still have fun programming computers, 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 very useful machines. The good news is that learning to program a computer is easy. If you can read this book, you can program a computer. A computer **program** is just a bunch of instructions run by a computer, just like a storybook is just a whole bunch of sentences read by the reader.

These instructions are like the turn-by-turn instructions you might get for walking to a friend's house. (Turn left at the light, walk two blocks, keep walking until you find the first blue house on the right.) The computer follows each instruction that you give it in the order that you give it. Video games are themselves nothing but computer programs. (And very

page, then look for the file called **Python 3.1 Windows Installer** (Windows binary -- does not include source) and click on its link to download Python for Windows.

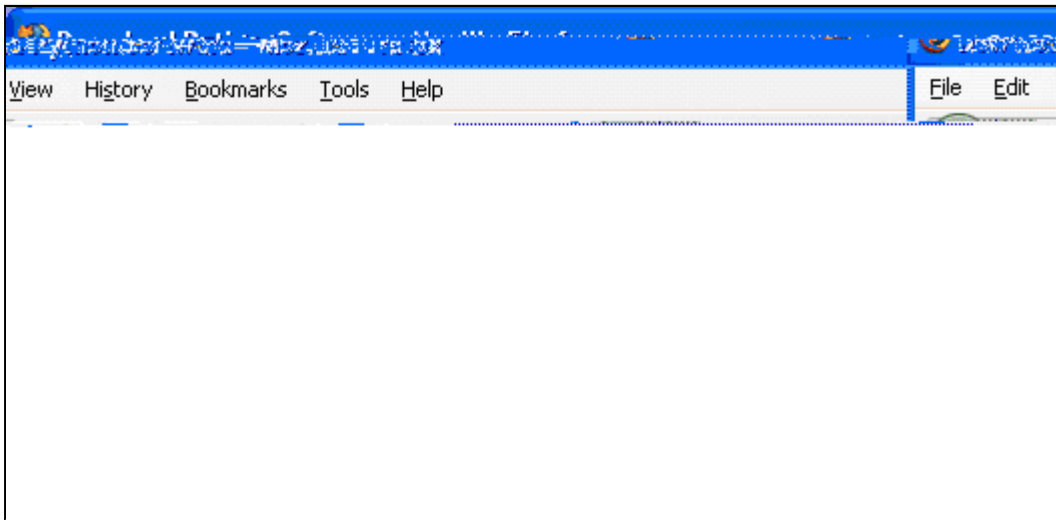


Figure 1-1: Click the Windows installer link to download Python for Windows from <http://www.python.org>

Double-click on the *python-3.1.msi* file that you've just downloaded to start the Python installer. (If it doesn't start, try right-clicking the file and choosing Install.) Once the installer starts up, click the **Next** button and just accept the choices in the installer as you go (no need to make any changes). When the install is finished, click **Finish**.

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.

The installation for Mac OS is similar. Instead of downloading the .msi file from the Python website, download the .dmg Mac Installer Disk Image file instead. The link to this file will look something like "Mac Installer disk image (3.1.1)" on the "Download Python Software" web page.

If your operating system is Ubuntu, you can install Python by opening a terminal window (click on Applications > Accessories > Terminal) and entering `sudo apt-get install python3` then pressing Enter. You will need to enter the root password to install Python, so ask the person who owns the computer to type in this password.

There may be a newer version of Python available than 3.1. If so, then just download the latest version. The game programs in this book will work just the same. If you have any problems, you can always Google for "installing Python on <your operating system's name>". Python is a very popular language, so you should have no difficulty finding help.

A video tutorial of how to install Python is available from this book's website at <http://inventwithpython.com/videos/>.

Starting Python

If your operating system is Windows XP, you should be able to run Python by choosing **Start > Programs > Python 3.1 > IDLE** (Python GUI). When it's running it should look something like Figure 1-2. (But different operating systems will look slightly different.)

Figure 1-2: The IDLE program's interactive shell on Windows.

IDLE stands for **I**nteractive **D**evelopment **E**nvironment. The development environment is software that makes it easy to write Python programs. We will be using IDLE to type in our programs and run them.


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

Text Wrapping in This Book

Some lines of code are too long to fit on one line on the page, and the text of the code will

will

tutorial of how to use the diff tool is available from this book's website at <http://inventwithpython.com/videos/>.

Summary

This chapter has helped you get started with the Python software by showing you the python.org website where you can download it for free. After installing and starting the Python IDLE software, we will be ready to learn programming starting in the next chapter.

This book's website at <http://inventwithpython.com> has more information on each of the chapters, including an online tracing website that can help you understand what exactly each line of the programs do.

Figure 2-1: Type 2+2 into the shell.

As you can see, we can use the Python shell just like a calculator. This isn't a program by itself because we are just learning the basics right now. The + sign tells the computer to add the numbers 2 and 2. To subtract numbers use the - sign, and to multiply numbers use an asterisk (*), like so:

When used in this way, +, -, *, and / are called **operators**

Expressions

This is like how a cat is a type of pet, but not all pets are cats. Someone could have a pet dog or a pet lizard. An **expression** is made up of values (such as integers like 8 and 6) connected by an operator (such as the * multiplication sign). A single value by itself is also considered an expression.

Storing Values in Variables



evaluate expressions (that is, reduce the expression to a single value), and that expressions are values (such as 2 or 5) combined with operators (such as + or -). You have also learned that you can store values inside of variables in order to use them later on.

In the next chapter, we will go over some more basic concepts, and then you will be ready to program!



Topics Covered In This Chapter:

- Flow of execution
- Strings
- String concatenation
- Data types (such as strings or integers)
- Using IDLE to write source code.
- Saving and running programs in IDLE.
- The `print()` function.
- The `input()` function.
- Comments
- Capitalizing variables
- Case-sensitivity
- Overwriting variables

That's enough of integers and math for now. Python is more than just a calculator. Now let's see what Python can do with text. In this chapter, we will learn how to store text in variables, combine text together, and display them on the screen. Many of our programs will use text to display our games to the player, and the player will enter text into our programs through the keyboard. We will also make our first program, which greets the user with the text, "Hello World!" and asks for the user's name.

Strings

In Python, we work with little chunks of text called **strings**. We can store string values inside variables just like we can store number values inside variables. When we type

strings, we put them in between two single quotes ('), like this:



Saving Your Program

A video tutorial of how to use the file editor is available from this book's website at <http://inventwithpython.com/videos/>.

If you get an error 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
```

...then this means you are running the program with Python 2, instead of Python 3. You

lists the differences between Python 2 and 3 that you will need for this book.

Opening The Programs You've Saved

To load a saved program, choose **File > Open**. Do that now, and in the window that appears choose *hello.py* and press the **Open** button. Your saved *hello.py* program should open in the File Editor window.

Now it's time to run our program. From the File menu, choose **Run > Run Module** or just press the F5 key on your keyboard. Your program should run in the shell window that appeared when you first started IDLE. Remember, you have to press F5 from the file editor's window, not the interactive shell's window.

When your program asks for your name, go ahead and enter it as shown in Figure 3-5:

Figure 3-5: What the interactive shell looks like when running the "Hello World" program.

Now, when you push Enter, the program should greet you (the **user**) by name.

The

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

Variable Names

The computer doesn't care what you name your variables, but you should. Giving variables names that reflect what type of data they contain makes it easier to understand what a program does. Instead of name

the screen when the `print ()` function is executed.

Strings are just a different data type that we can use in our programs. We can use the `+` operator to concatenate strings together. Using the `+` operator to concatenate two strings together to form a new string is just like using the `+` operator to add two integers to form a new integer (the sum).

In the next chapter, we will learn more about variables so that our program will remember the text and numbers that the player enters into the program. Once we have learned how to use text, numbers, and variables, we will be ready to start creating games.



Topics Covered In This Chapter:

- `import` statements
- Modules
- Arguments
- `while` statements
- Conditions
- Blocks
- Booleans
- Comparison operators
- The difference between `=` and `==`.
- `if` statements
- The `break` keyword.
- The `str()` and `int()` functions.
- The `random.randint()` function.

The "Guess the Number" Game

We are going to make a "Guess the Number" game. In this game, the computer will think of a random number from 1 to 20, and ask you to guess the number. You only get six guesses, but the computer will tell you if your guess is too high or too low. If you guess the number within six tries, you win.

This is a good game for you to start with because it uses random numbers, loops, and input from the user in a fairly short program. As you write this game, you will learn how to convert values to different data types (and why you would need to do this).

This line creates a new variable named `guessesTaken`

because each time the `randint()` function is called, it returns some random number, just like when you roll dice you will get a random number each time.

```
>>> import random
```

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. 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.

Calling Functions that are Inside Modules

By the way, be sure to enter `random.randint(1, 20)` and not just `randint(1, 20)`, or the computer will not know to look in the `random` module for the `randint()` o t2(+)-248d(u)-2

Loops

Line 12 has something called a `while` statement, which indicates the beginning of a while loop. **Loops** are parts of code that are executed over and over again. But before we can learn about `while` loops, we need to learn a few other concepts first. Those concepts are blocks, booleans, comparison operators, conditions, and finally, the `while` statement.

Blocks

A **block** is one or more lines of code grouped together with the same minimum amount of indentation. You can tell where a block begins and ends by looking at the line's **indentation** (that is, the number of spaces in front of the line).

A block begins when a line is indented by four spaces. Any following line that is also indented by four spaces is part of the block. A block within a block begins when a line is indented with another four spaces (for a total of eight spaces in front of the line). The block ends when there is a line of code with the same indentation before the block started.

Below is a diagram of the code with the blocks outlined and numbered. The spaces have black squares filled in to make them easier to count.

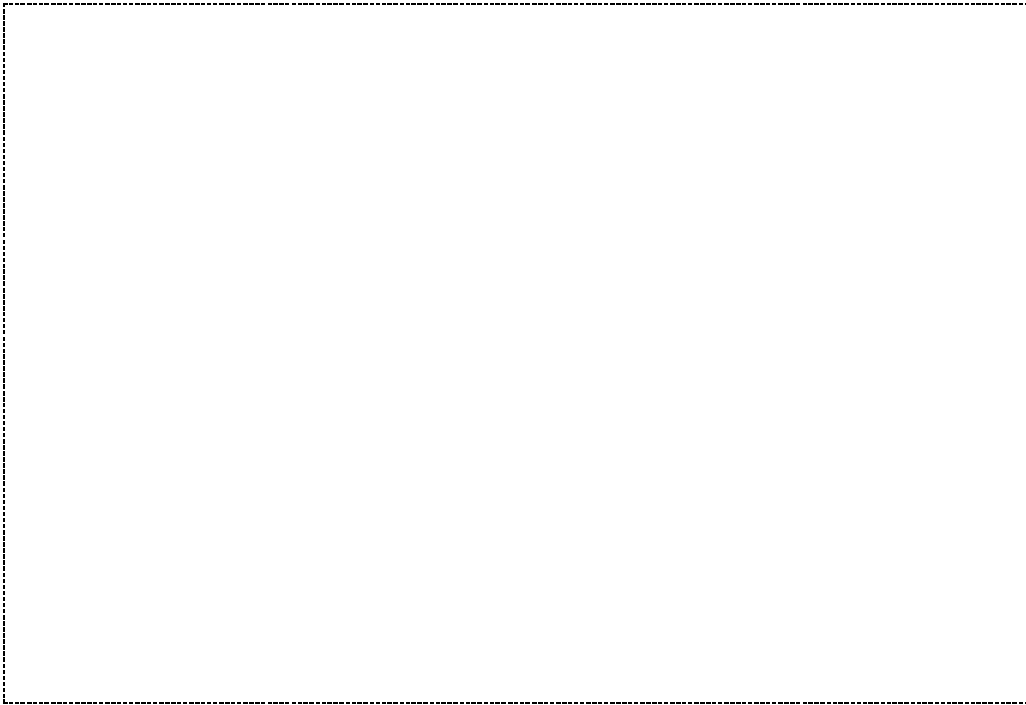
Figure 4-1: Blocks and their indentation. The black dots represent spaces.

For example, look at the code above. The spaces have been replaced with dark squares to make them easier to count. Line 12 has an indentation of zero spaces and is not inside any block. Line 13 has an indentation of four spaces and is inside a block.

Block 1 starts on line 13 and ends on line 15. Line 16 is not inside any block.

False

The condi



True

Looping with While Statements

The `while` statement marks the beginning of a loop. Sometimes in our programs, we want the program to do something over and over again. 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 `while`-block. (In our 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 our program, the first line after the `while`-block is line 28.)

```
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 `low[3](e)aken < 6:`

`int('forty-two')` also produces an error. That said, the `int()` function is slightly forgiving- if our string has spaces on either side, it will still run without error. This is why the `int(' 42 ')` call works.

The `3 + int('2')` line shows an expression that adds an integer 3 to the return value of `int('2')` (which evaluates to 2 as well). The expression evaluates to `3 + 2`, which then evaluates to 5. So even though we cannot add an integer and a string (`3 + '2'` would show us an error), we can add an integer to a string that has been converted to an integer.

Remember, back in our program on line 15 the `guess` variable originally held the string value of what the player typed. We will overwrite the string value stored in `guess` with the integer value returned by the `int()` function. This is because we will later compare the player's guess with the random number the computer came up with. We can only compare two integer values to see if one is greater (that is, higher) or less (that is, lower) than the other. We cannot compare a string value with an integer value to see if one is greater or less than the other, even if that string value is numeric such as `'5'`.

In our Guess the Number game, if the player types in something that is not a number, then the function call `int()` will result in an error and the program will crash. In the other games in this book, we will add some more code to check for error conditions like this and give the player another chance to enter a correct response.

Notice that calling `int(guess)` does not 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. We must assign this return value to `guess` in order to change the value in `guess` to an integer with this full line: `guess = int(guess)`

Incrementing Variables

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

Once the player has taken a guess, we want to increase the number of guesses that we remember the player taking.

The first time that we enter the loop block, `guessesTaken` has the value of 0. Python will take this value and add 1 to it. `0 + 1` is 1. Then Python will store the new value of 1 to `guessesTaken`.

Think of line 17 as meaning, "the `guessesTaken` variable should be one more than what it already is".

When we add 1 to an integer value, programmers say they are **incrementing** the value (because it is increasing by one). When we subtract one from a value, we are **decrementing** the value (because it is decreasing by one). The next time the loop block

loops around, `guessesTaken` will have the value of 1 and will be incremented to the value 2.

Is the Player's Guess Too Low?

Lines 19 and 20 check if the number that the player guessed is less than the secret random number that the computer came up with. If so, then we want to tell the player that their guess was too low by printing this message to the screen.

if Statements

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

Line 19 begins an `if` statement with the keyword, `if`. Next to the `if` keyword is the condition. Line 20 starts a new block (you can tell because the indentation has increased from line 19 to line 20.) The block that follows the `if` keyword is called an `if`-block. An `if` statement is used if you only want a bit of code to execute if some condition is true. Line 19 has an `if` statement with the condition `guess < number`. If the condition evaluates to `True`, then the code in the `if`-block is executed. If the condition is `False`, then the code in the `if`-block is skipped.

Figure 4

If the integer the player enters is less than the random integer the computer thought up, the program displays `Your guess is too low`. If the integer the player enters is equal to or larger than the random integer (in which case, the condition next to the `if` keyword would have been `False`), then this block would have been skipped over.

In Line 32, we use the comparison operator `!=` with the `if` statement's condition to mean "is not equal to." If the value of the player's guess is lower or higher than (and therefore, not equal to) the number chosen by the computer, then this condition evaluates to `True`, and we enter the block that follows this `if` statement on line 33.

Lines 33 and 34 are inside the `if`-block, and only execute if the condition is `True`.

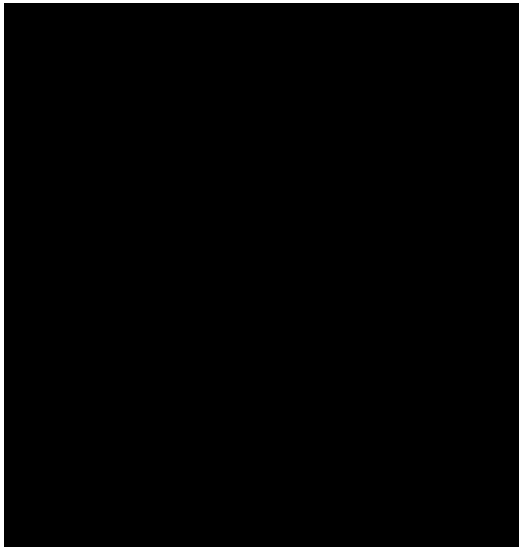
```
33.     number = str(number)
34.     print('Nope. The number I was thinking of was ' +
           number)
```

kberifes the(he)(ow)-6(va)4(l)-2(ue)4(of)-7()]TJ /T1_1 12 Tf6(kberif)Tj /T1_0 12 Tf (.)T-21113.32 -
ba(e)4(s)-8(e)-d on-nd8-2(t)-4(s)-1(i)-n1(s)-(ut)-4(r)u(c)t(l)-1(

Figure 4-4: The tracing web page.

The left side of the web page shows the source code, and the highlighted line is the line of code that is about to be executed. You execute this line and move to the next line by clicking the "Next" button. You can also go back a step by clicking the "Previous" button, or jump directly to a step by typing it in the white box and clicking the "Jump" button.

On the right side of the web page, there are three sections. The "Current variable values" section shows you each variable thatr



Topics Covered In This Chapter:

- Using `print()`'s end keyword argument to skip newlines.
- Escape characters.
- Using single quotes and double quotes for strings.

Make 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 and entered to the computer. The output is the text displayed on the screen. In Python, the `print()` function can be used for displaying textual output on the screen. We've learned how the basics of using the `print()` function, but there is more to learn about how strings and `print()` work in Python.

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!
```

Joke's Source Code

Here is the source code for our short jokes program. Type it into the file editor and save it as *jokes.py*. If you do not want to type this code in, you can also download the source code from this book's website at the URL <http://inventwithpython.com/chapter5>.

Important Note! Be sure to run this program with 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. You can click on Help and then About IDLE to find out what version of Python you have.

jokes.py

This code can be downloaded from <http://inventwithpython.com/jokes.py>
If you get errors after typing this code in, compare it to the book's code with the online diff tool at <http://inventwithpython.com/diff> or email the author at al@inventwithpython.com

```
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!')
```

Don't worry if you don't understand everything in the program. Just save and run the program. Remember, if your program has bugs in it, you can use the online diff tool at <http://inventwithpython.com/chapter5>.

How the Code Works

Let's look at the code more

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

Here we have three `print()` function calls. Because we don't want to tell the player what the joke's punch line is, we have a call to the `input()` function after the first `print()`. The player can read the first line, press Enter, and then read the punch line.

The user can still type in a string and hit Enter, but because we aren't storing this string 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 can be useful to keep our text from being bunched up together.

Escape Characters

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

In the first `print()` above, you'll notice that we have a slash right before the single quote (that is, the apostrophe). This backslash (`\` is a backslash, `/` is a forward slash) tells us that the letter right after it is an **escape character**. An escape character helps us print out letters that are hard to enter into the source code. There are several different escape characters, but in our call to `print()` the escape character is the single quote.

We have to have the single quote escape character because otherwise the Python interpreter would think that this quote meant the end of the string. But we want this quote to be a part of the string. When we print this string, the backslash will not show up.

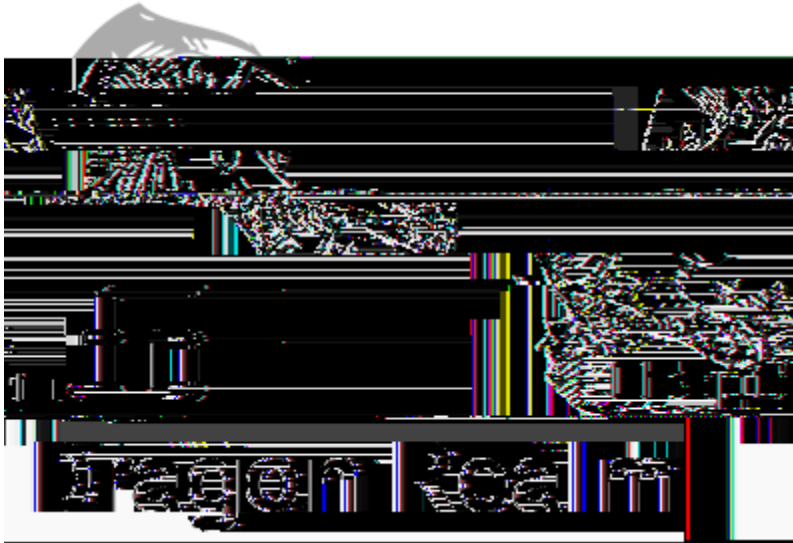
Some Other Escape Characters

What if you really want to display a backslash? This line of code would not work:

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

That `print()` function call would show up as:

to the previous line, instead



Topics Covered In This Chapter:

- The `time` module.
- The `time.sleep()` function.
- The `return` keyword.
- Creating our own functions with the `def` keyword.
- The `and` and `or` and `not` boolean operators.
- Truth tables
- Variable scope (Global and Local)
- Parameters and Arguments
- Flow charts

Introducing Functions

We've already used two functions in our previous programs: `input()` and `print()`. In our previous programs, we have called these functions to execute the code that is inside these functions. In this chapter, we will write our own functions for our programs to call. A function is like a mini-program that is inside of our program. Many times in a program we want to run the exact same code multiple times. Instead of typing out this code several times, we can put that code inside a function and call the function several times. This has the added benefit that if we make a mistake, we only have one place in the code to change it.

The game we will create to introduce functions is called "Dragon Realm", and lets the player make a guess between two caves which randomly hold 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 will share their treasure with you. Other dragons are greedy and hungry, and will 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 is given a choice between the two.

Open a new file editor window by clicking on the **File** menu, then click on **New Window**. In the blank window that appears type in the source code and save the source code 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
```

Dragon Realm's Source Code

Here is the source code for the Dragon Realm game. Typing in the source code is a great way to get used to the code. But if you don't want to do all this typing, you can download the source code from this book's website at the URL <http://inventwithpython.com/chapter6>. There are instructions on the website that will tell you how to download and open the source code file. You can use the online diff tool on the website to check f1.4 TD con/ia4(khe)-6ns

Important Note! Be sure to run this program with 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. You can click on Help and then About IDLE to find out what version of Python you have.

dragon.py

This code can be downloaded from <http://inventwithpython.com/dragon.py>
If you get errors after typing this code in, compare it to the book's code with the online diff tool at <http://inventwithpython.com/diff> or email the author at al@inventwithpython.com

```
1. import random
2. import time
3.
4. def displayIntro():
5.     print('You are on a planet 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()
```



```
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
```

Here we have two `import` statements. We import the `random` module like we did in the Guess the Number game. In Dragon Realm, we will also want some time-related functions that the `time` module includes, so we will import that as well.

Defining the `displayIntro()` Function

```
4. def displayIntro():  
5.     print('You are on a planet full of dragons. In front
```

def Statements

Try typing the following into the interactive shell:

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

Experimenting with the `not` Operator

The third boolean operator is `not`. The `not` operator is different from every other operator we've seen before, because it only works on one value, not two. There is only one value on the right side of the `not`

program execution will continue on past the `while` loop.

The reason we have a loop here is because the player may have typed in 3 or 4 or HELLO. Our program doesn't make sense of this, so if the player did not enter 1 or 2, then the program loops back and asks the player again. In fact, the computer will patiently ask the player for the cave number over and over again until the player types in 1 or 2. When the player does that, the while-block's condition will be `False`, and we will jump down past the while-block and continue with the program.

Return Values

```
17.     return cave
```

This is the **return** keyword, which only appears inside `def`-blocks. Remember how the `input()` function returns the string value that the player typed in? Or how the `randint()` function will return a random integer value? Our function will also return a value. It returns the string that is stored in `cave`.

This means that if we had a line of code like `spam = chooseCave()`, the code inside `chooseCave()` would be executed and the function call will evaluate to `chooseCave()`'s return value.

we create a variable named `spam` inside of the function, the Python interpreter will consider them to be two separate variables. That means we can change the value of `spam` inside the function, and this will not change the `spam` variable that is outside of the


```
# The global variable was not changed in funky():
```

Parameters are local variables that get defined when a function is called. The value stored in the parameter is the argument that was passed in the function call.

Parameters

For example, here is a short program that demonstrates parameters. Imagine we had a short program that looked like this:

```
print('Hello, ' + fizzy)

print('Say hello to Alice.')
fizzy = 'Alice'
sayHello()
print('Do not forget to say hello to Bob.')
sayHello()
```

When we run this code, it looks like this:

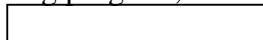
```
Say hello to Alice.
Hello, Alice
Do not forget to say hello to Bob.
Hello, Alice
```

This program's `sayHello()` function does not have a parameter, but uses the global variable `fizzy` directly. Remember that you can read global variables inside of functions, you just can't modify the value stored in the variable.

Without parameters, we have to remember to set the `fizzy` variable before calling `sayHello()`. In this program, we forgot to do so, so the second time we called `sayHello()` the value of `fizzy` was still `'Alice'`. Using parameters makes function calling simpler to do, especially when our programs are very big and have many functions.

Local Variables and Global Variables with the Same Name

Now look at the following program, which is a bit different. To make it clear to see, the




```
NameError: name 'sayGoodBye' is not defined
```

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

```
def sayGoodBye():  
    print('Good bye!')
```


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.

The Colon :

You may have noticed that we always place a colon at the end of `if`, `else`, `while`, and `def` statements. The colon marks the end of the statement, and tells us that the next line should be the beginning of a new block.

Where the Program Really Begins

```
35. playAgain = 'yes'
```

function lets the player type in the cave they choose to go into. When the return cave line in this function executes, the program execution jumps back down here, and the local variable `cave`'s value is the return value of this function. The return value is stored in a new variable named `caveNumber`

We went through the source code from top to bottom. If you would like to go through the

carefully thought about what exactly the program is doing. There are three types of bugs

Figure 7-1: The Debug Control window.

Now when you run the Dragon Realm game (by pressing F5 or clicking **Run**, then **Run Module**

you have checked the **Source** checkbox in the Debug Control window), the first line of code is highlighted in gray. Also, the Debug Control window shows that you are on line 1, which is the `import random` line.

The debugger lets you execute one line or code at a time (called "stepping"). To execute a single instruction, click the **Step** button in the Debug Window. Go ahead and click the Step button once. This will cause the Python interpreter to execute the `import random` instruction, and then stop before it executes the next instruction. The Debug Control window will change to show that you are now on line 2, the `import time` line.

Stepping

Stepping is the process of executing one instruction of the program at a time. Doing this lets you see what happens after running a single line of code, which can help you figure out where a bug first appears in your programs.

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. This window will also tell you what line number it is on and show you the instruction itself.

Click the Step button again to run the `import time` instruction. The debugger will execute this `import` statement and then move to line 4. The debugger skipped line 3 because it is a blank line. Notice that you can only step forward with the debugger, you cannot go backwards.

Click the Step button three more times. This will execute the three `def` statements to define the functions. The debugger skips over the `def`-blocks of these functions because we are only defining the functions, not calling them. As you define these functions, they will

stored. Global variables are the variables that are created outside of any functions (that is, in the global scope). There is also a **Local 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. Since we are still in the global scope, this area is blank.

The Python debugger (and almost all debuggers) only lets you step forward in your program. Oogra yo8(ha)-vexe4(u1(t)-2(e)-d1(a)41(i)-2(ns)-t(i)-2(r)uc(a)4(t)-8(i)-2(h(yoc(a)(ha)-nnot(1

Figure 7-3: Keep stepping until you reach line 38.


```
number2))
```

Type the program in exactly as it is above, even if you can already tell what the bug is. Then try running the program by pressing F5. This is a simple arithmetic game 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 not right! This program has a semantic bug in it. Even if the user types in the correct answer, the program says they are wrong.

You could look at the code and think hard about where it went wrong. That works sometimes. But you might figure out the cause of the bug quicker if you run the program under the debugger. At the top of the interactive shell window, click on **Debug**, then **Debugger** (if there is no check already by the Debugger menu item) to display the Debug Control window. In the Debug Control window, make sure the all four checkboxes (Stack, Source, Locals, Globals) are checked. This makes the Debug Control window provide the most information. Then press **F5** in the file editor window to run the program under the debugger.

The debugger starts at the `import random` line. Nothing special happens here, so just click **Step** to execute it. You should see the `random` module at the bottom of the Debug Control window in the Globals area.

Click **Step** again to run line 2. A new file editor window will pop open. Remember that the `randint()` function is inside the `random` module. When you stepped into the function, you stepped into the `random` module because that is where the `randint` function is. The functions that come with Python's modules almost never have bugs in their code, so you can just click **Out** to step out of the `randint()` function and back to your program. After you have stepped out, you can close the `random` module's window.

Line 3 is also a call to the `randint()` function. We don't need to step through this code, so just click **Over** to step over this function call. The `randint()` function's code is still executed, it is just executed all at once so that we don't have to step through it.

Line 4 is a `print()` call to show the player the random numbers. But since we are using the debugger, we 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. When I ran the debugger, it looked like this:

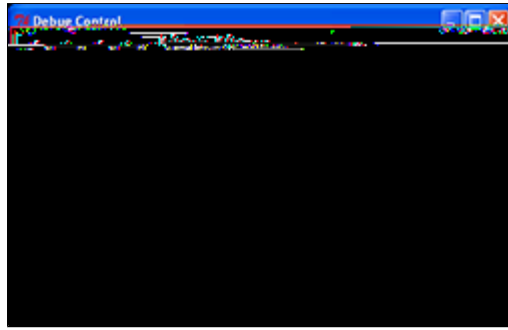


Figure 7-4: The Debug Control window.

The `number1` variable has the value 9 and the `number2` variable has the value 10. When you click Step, the program will display the string in the `print()` call with these values. (Of course, we use the `str()` function so that we can concatenate the string version of these integers.)

Clicking on Step on line 5 will cause the debugger to wait until the player enters a response. Go ahead and type in the correct answer (in my case, 19) into the interactive shell window. The debugger will resume and move down to line 6.

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.

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 the `answer == number1 + number2` condition would have evaluated to `'19' == 19`. A string value and an integer value will always not equal each other, so the condition would have evaluated to `False`.

That is the bug in the program. The bug is that we use `answer` when we should be using `int(answer)`. Go ahead and change line 6 to use `int(answer) == number1 + number2` instead of `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 to make sure the program doesn't tell us we gave the correct answer. We have now debugged this program. Remember, the computer will run your programs exactly as you type them, even if what you type is not what you intend.

Break Points

Stepping through the code one line at a time might still be too slow. Often you will want the program to run at normal speed until it reaches a certain line. You can do this with break points. A **break point** is set on a line when you want the debugger to take control once execution reaches that line. So if you think there is a problem with your code on, say, line 17, just set a break point on line 17 and when execution reaches that line, the debugger will stop execution. Then you can step through a few lines to see what is happening. Then you can click Go to let the program execute until it reaches the end (or another break point).

To set a break point, right-click on the line that you want a break point on and select "Set Breakpoint" from the menu that appears. The line will be highlighted with yellow to indicate a break point is on that line. 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 of Using Break Points

Let's try debugging a program with break points. Here is a program that



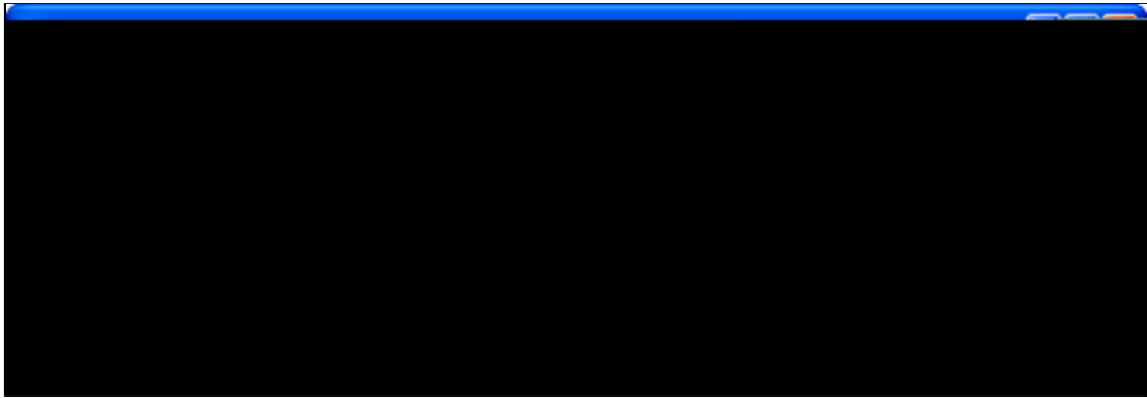


Figure 7-6: Three break points set.

After setting the breakpoints, click **Go** in the Debug Control window. The program will run at its normal speed until it reaches `flip 100`. On that flip, the condition for the `if` statement on line 13 is `True`. This causes line 14 (where we have 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. You can click **Go** one more time to continue the execution until it reaches the next break point.

And if you click **Go** again, the execution will continue until the next break point is reached, which is on line 12. You probably noticed that the `print()` functions on lines 12, 14 and 16 are called in a different order than they appear in the source code. That is because they are called in the order that their `if` statement's condition becomes `True`. Using the debugger can help make it clear why this is.

Summary

Writing programs is only part of the work for making games. The next part is making sure the code we wrote actually works. Debuggers let us step through the code one line at a time, while examining which lines execute (and in what order) and what values the variables contain. When this is too slow, we can set break points and click **Go** to let the program run normally until it reaches a break point.

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

we

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!

Designing a Program with a Flowchart

This game is a bit more complicated than the ones we've seen so far, so let's take a moment to think about how it's put together. First we'll create a flow chart (like the one at the end of the Dragn(r) 0 0 8(')-6rsoe endo oge(r) p u iv6(m)-2(e)4un(r) 6(m)-zve ssre(t)8(t)-2(o i)-2(tI)o1

that we hadn't considered

Now let's think about what happens when we play

Branching from a Flowchart Box

There are two possibilities: the letter will either be in the word or it won't be. This means we need to add *two* new boxes to our flowchart. From the "Ask player to guess a letter" box, we can only move to the "Letter is in secret word" box *or* the "Letter is not in secret word" box. This will create a branch (that is, a split) in the flow chart, as show in Figure 8-4:

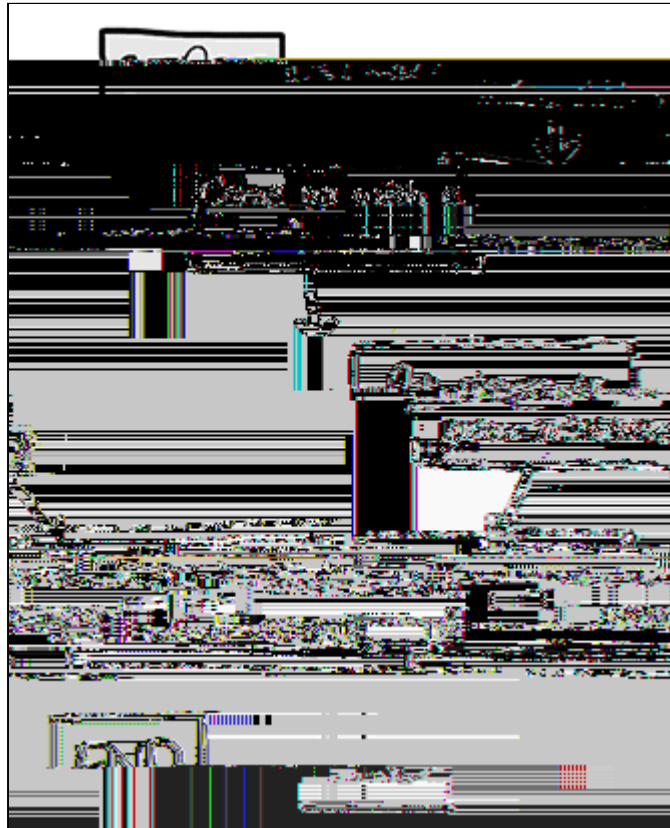


Figure 8-4: There are two different things that could happen after the player guesses, so have two arrows going to separate boxes.

If the letter is in the secret word, we need to check to see if the player has guessed all the letters, which would mean they've won the game. But, if the letter is not in the secret word, another body part is added to the hanging man.

We can add boxes for those cases too. We don't need an arrow from the "Letter is in

Guessing Again

This flow chart might look like it is finished, but there's something we're forgetting: the player doesn't guess a letter just once. They have to keep guessing letters over and over until they either win or lose. We need to draw two new arrows so the flow chart shows this, as shown in Figure 8-7.

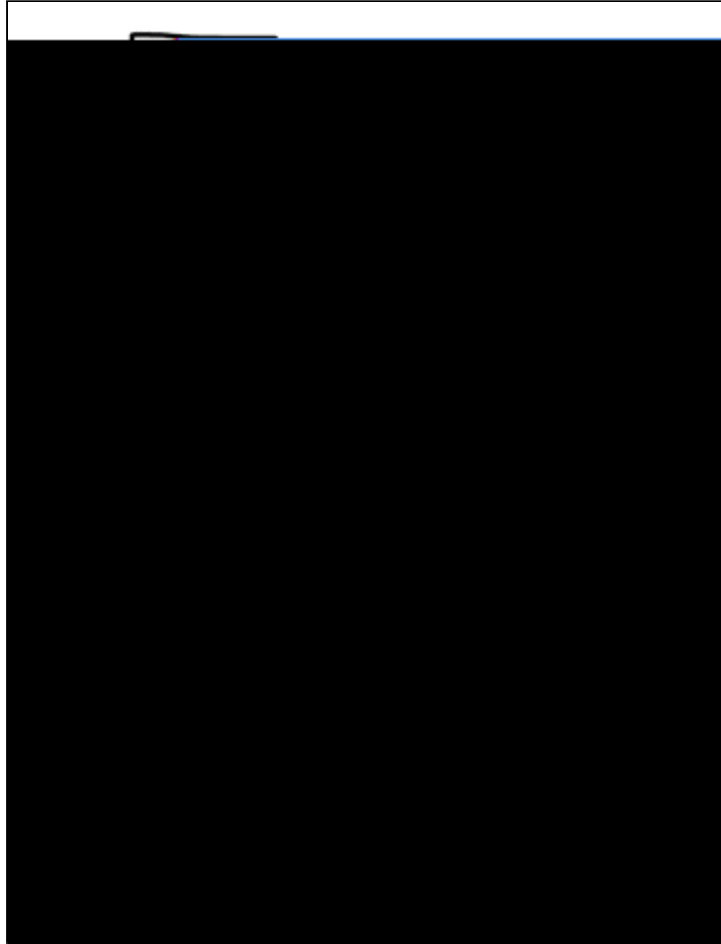


Figure 8-7: The game does not always end after a guess. The new arrows (outlined) show that the player can guess again.

We are forgetting something else, as well. What if the player guesses a letter that they've guessed before? Rather than have them win or lose in this case, we'll allow them to guess a different letter instead, as shown in Figure 8-8.

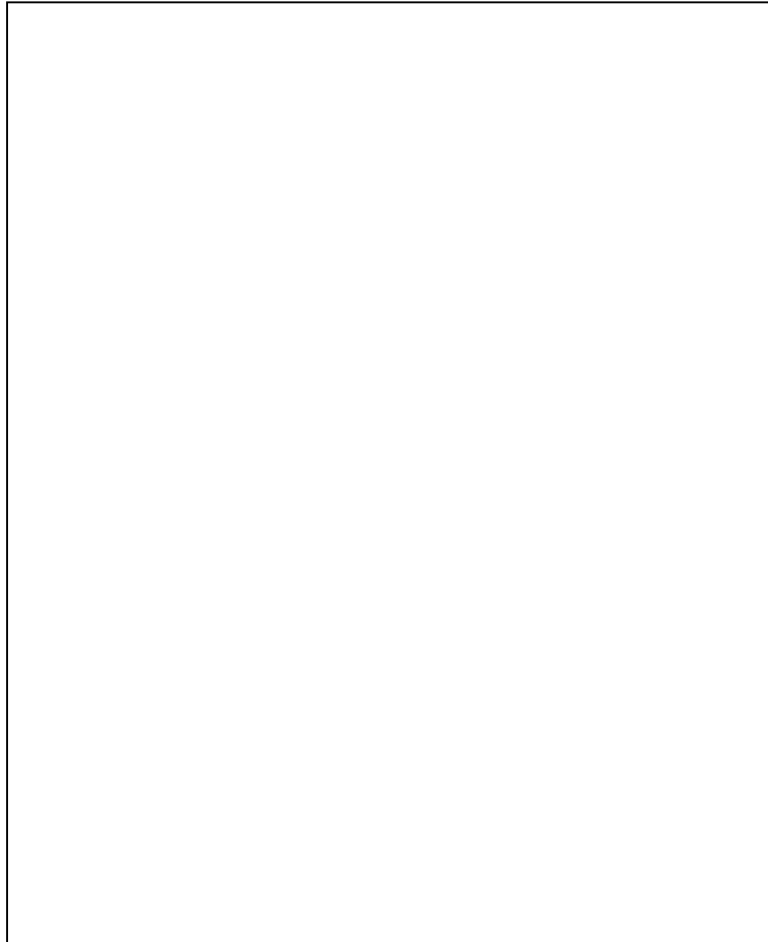


Figure 8-8: Adding a step in case the player guesses a letter they already guessed.

Offering Feedback to the Player

We also need some way to show the player how they're doing. In order to do this, we'll show them the hangman board, as well as 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.

We'll need to update this information every time the player guesses a letter. We can add a "Show the board and blanks to the player." box to the flow chart between the "Come up with a secret word" box and the "Ask player to guess a letter" box, as shown in Figure 8-9. This box will remind us that we need to show the player an updated hangman board so they can see which letters they have guessed correctly and which letters are not in the secret word.

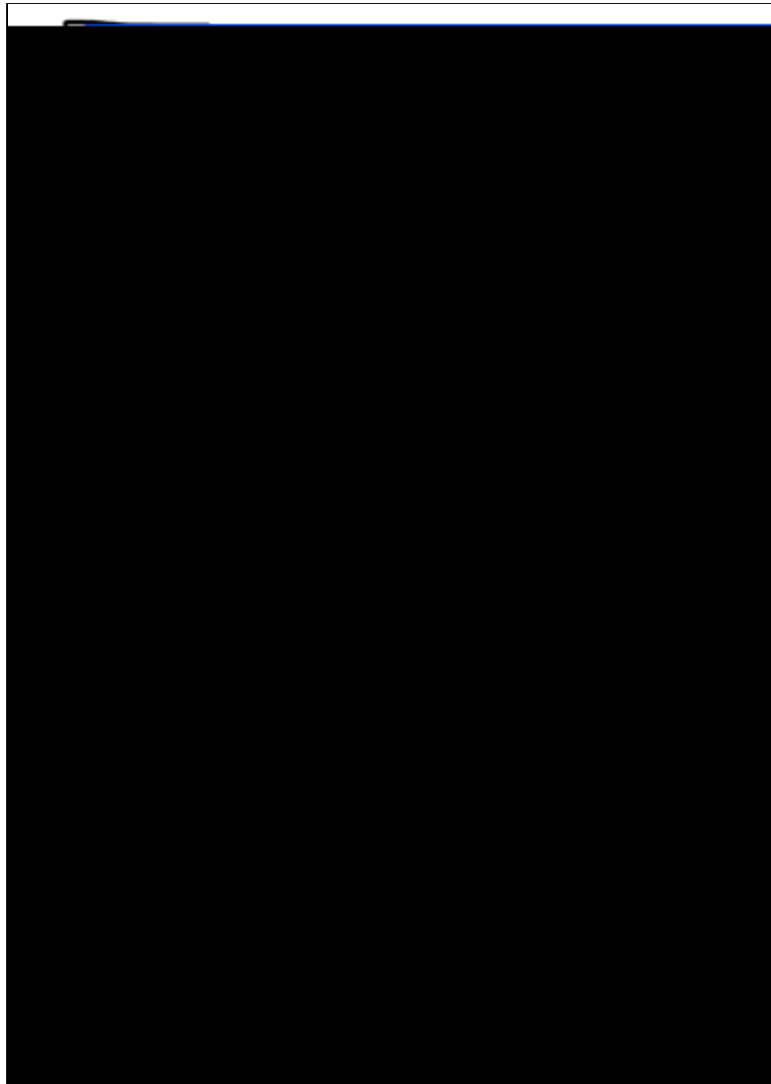


Figure 8-9: Adding "Show the board and blanks to the player." to give the player feedback.

That looks good! This flow chart completely maps out everything that can possibly happen in Hangman, and in what order. Of course this flow chart is just an example-you won't really need to use it, because you're just using the source code that's given here. But when you design your own games, a flow chart can help you remember everything you need to code.

Summary: The Importance of Planning Out the Game

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 that you create bugs by changing too little or too much. It is much better to know what you want to build before you build it.



Topics Covered In This Chapter:

Methods

The `append()` list method

The `lower()` and `upper()` string methods

The `reverse()` list method

The `split()` string method

The `range()` function

The `list()` function

`for` loops

`elif` statements

The `startswith()` and `endswith()` string methods.

The dictionary data type.

key-value pairs

The `keys()` and `values()` dictionary methods

Multiple variable assignment, such as `a, b, c = [1, 2, 3]`

This game introduces many new concepts. But don't worry; we'll experiment with these programming concepts in the interactive shell first. Some data types such as strings and lists have functions that are associated with their values called methods. We will learn several different methods that can manipulate strings and lists for us. We will also learn about a new type of loop called a `for` loop and a new type of data type called a dictionary. Once you understand these concepts, it will be much easier to understand the game in this chapter: Hangman.

You can learn more from Wikipedia: [http://en.wikipedia.org/wiki/Hangman_\(game\)](http://en.wikipedia.org/wiki/Hangman_(game))


```

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 = getG-2(e)-2(s)-2(s)-2( )-2(=2(g)-2(e)-2(t)-2(G-2(S)-2(,)-2P2(d)

```

```

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 is going to randomly select a secret word from a list of secret words. This means we will need the random module imported.

```

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

```



```
['apples', 'oranges', 'HELLO WORLD']
```

Because `animals[0]` evaluates to the string `'aardvark'` and `animals[2]` evaluates to the string `'antelope'`, then the expression `animals[0] + animals[2]` is the same as `'aardvark' + 'antelope'`. This string concatenation evaluates to `'aardvarkantelope'`.

'Alice said hello to Bob.' into the shell. This expression will evaluate to True.

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

Removing Items from Lists with `del` Statements

You can remove items from a list with a `del` statement. ("del" is short for "delete.") Try creating a list of numbers by typing: `spam = [2, 4, 6, 8, 10]` and then `del spam[1]`. Type `spam` to view the list's contents:

```
>>> 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

themselves. The image is also flipped on its side to make it easier to read:

Figure 9-1: The indexes of a list of lists.

Methods

Methods are just like functions, but they are always attached to a value. For example, all string values have a `lower()` method, which returns a copy of the string value in lowercase. You cannot just call `lower()` by itself and you do not pass a string argument to `lower()` by itself (as in `lower('Hello')`). You must attach the method call to a specific string value using a period.

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

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

.....


```
>>> 'Hello world'.upper()
```



```
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()
```

As you can see, this line is just one very long string, full of words separated by spaces. And at the end of the string, we call the `split()` method. The `split()` method changes this long string into a list, with each word making up a single list item. The "split" occurs wherever a space occurs in the string. The reason we do it this way, instead of just writing out the list, is that it is easier for us to type as one long string. If we created it as a list to begin with, we would have to type: `['ant', 'baboon', 'badger', ...]` and so on, with quotes and commas for every single word.

For an example of how the `split()` string method works, try typing this into the shell:

```
>>> 'My very energetic mother just served us nine
pies'.split()
['My', 'very', 'energetic', 'mother', 'just',
'served', 'us', 'nine', 'pies']
>>>
```

The result is a list of nine strings, one string for each of the words in the original string. The spaces are dropped from the items in the list. Once we've called `split()`, the words list will contain all the possible secret words that can be chosen by the computer for our Hangman game. You can also add your own words to the string, or remove any you don't want to be in the game. Just make sure that the words are separated by spaces.

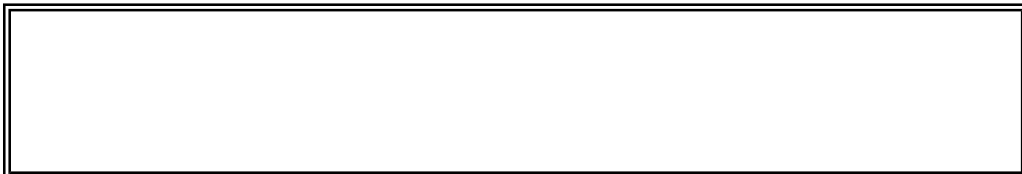
How the Code Works

Starting on line 61, we define a new function called `getRandomWord()`, which has a single parameter named `wordList`. We will call this function when we want to pick a single secret word from a list of secret words.


```
67.     print(HANGMANPICS[len(missedLetters)])
68.     print()
```

This code defines a new function named `displayBoard()`. This function has four parameters. This function will implement the code for the "Show the board and blanks to the player" box in our flow chart. Here is what each parameter means:

`HANGMANPICS` - This is a list of multi-line strings that will display the board as ASCII art. We will always pass the global `HANGMANPICS` variable as the argument



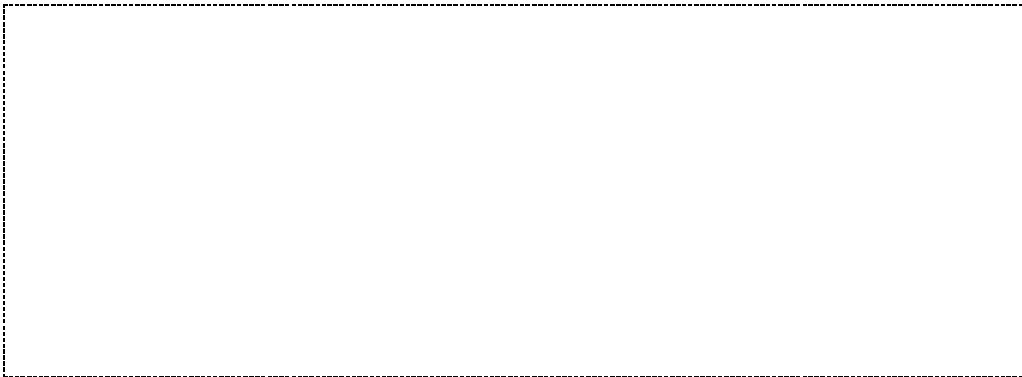
much like the `while` loops we have already seen).

for Loops

The `for`

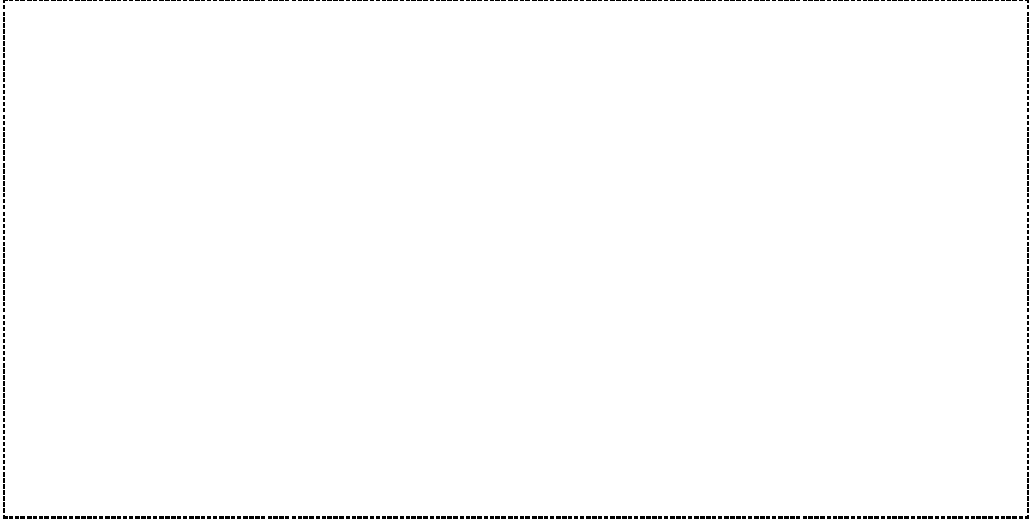
```
>>> for i in range([0, 1, 2, 3, 4, 5, 6, 7, 8,
9]):
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
>>>
```

Try typing this into the shell: `for thing in ['cats', 'pasta', 'programming', 'spam']:` and press Enter, then type `print('I really like ' + thing)` and press Enter, and then press Enter again to tell the shell to end the for-block. The output should look like this:



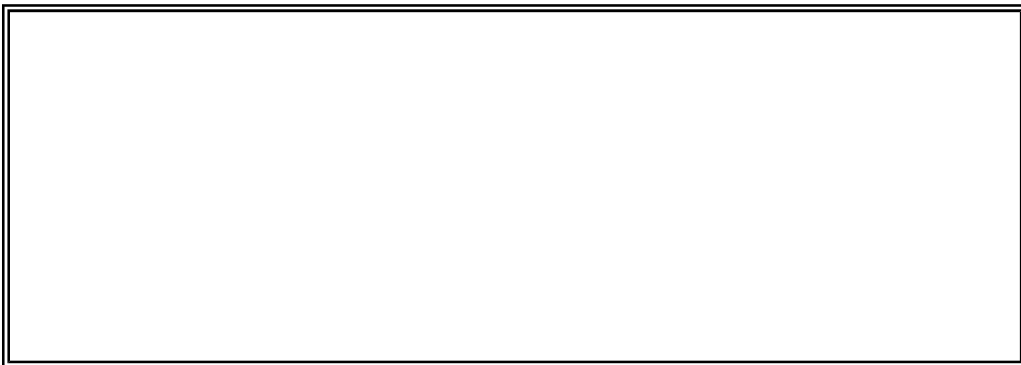
o
w
o
r
l
d
!
>>>

A



```
        print(letter, end=' ')
    print()
```

This `for` loop one line 71 will display all the missed guesses that the player has made. When you play Hangman on paper, you usually write down these letters off to the side so you know not to guess them again. On each iteration of the loop the value of `letter` will be each letter in



Get the Player's Guess

The `getGuess()` function we create next will be called whenever we want to let the player type in 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.
```

The `getGuess()` function has a string parameter called `alreadyGuessed` which contains the letters the player has already guessed, and will ask the player to guess a single letter. This single letter will be the return value for this function.

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

We will use a `while` loop because we want to keep asking the player for a letter until they enter text that is a single letter they have not guessed previously. Notice that 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 the entire function). Such a loop is called an **infinite loop**, because it will loop forever (unless it reaches a `break` statement).

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 converted to lowercase on line 90.

`elif` ("Else If") Statements

Take a look at the following code:

that her cat is fuzzy. If `catName` is anything else, then we tell the user her cat is not fuzzy.

But what if we wanted something else besides "fuzzy" and "not fuzzy"? We could put another `if` and `else` statement inside the first `else` block like this:

```
if catName == 'Fuzzball':  
    print('Your cat is fuzzy.')  
else:  
    if catName == 'Spots'
```

```
print('Your cat is neither fuzzy nor spotted  
nor fat nor puffy.')
```

If the condition for the `if` statement is `False`, then the program will check the condition for the first `elif` statement (which is `catName == 'Spots'`). If that condition is `False`, then the program will check the condition of the next `elif` statement. If *all* of the conditions for the `if` and `elif` statements are `False`, then the code in the `else` block executes.

But if one of the `elif` conditions are `True`, the `elif`-block code is executed and then execution jumps down to the first line

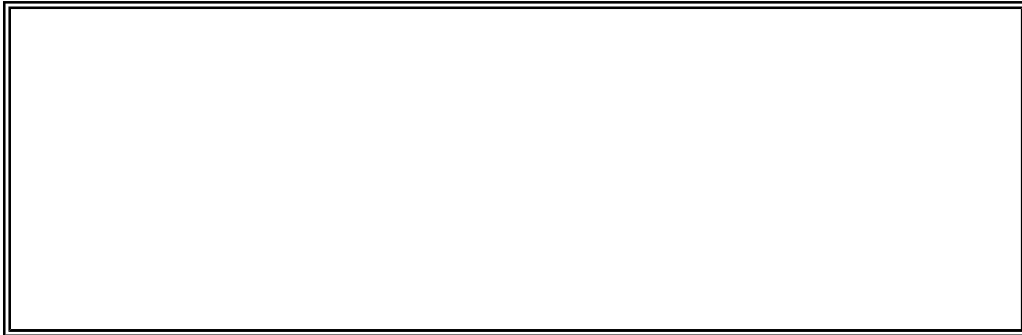


Figure 9-3 is an example of `elif` statements. Unless these three conditions are all `False`, the player will keep looping and keep being asked for a letter. But when all three of the conditions are `False`, then the `else`-block's `return` statement will run and we will exit this loop and function.

Figure 9-3: The `elif` statement.

Asking the Player to Play Again

```
100. def playAgain():
```



```
return 'yes'.startswith('y')  
  
return True
```

The point of the `playAgain()` function is to let the player type in yes or no to tell our program if they want to play another round of Hangman. 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 we have evaluated this expression! We can see that what this does is let the player type in a response, ~~we lower~~ lowercase the response, check if it a

doesn't.

We'll now start the code for the main part of the game, which will call the above functions as needed. Look back at our flow chart.



Figure 9-4: The complete flow chart of Hangman.

The Main Code for Hangman

We need to write code that does everything in this flow chart, and does it in order. The main part of the code starts at line 106. Everything previous was just function definitions and a very large variable assignment 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
```

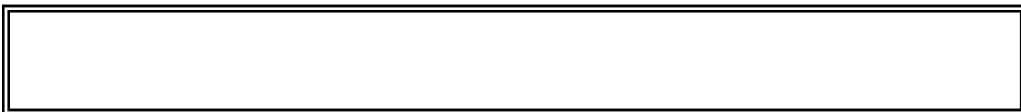
Line 106 is the first actual line that executes in our game. We start by assigning a blank string for `missedLetters` and `correctLetters`, because the player has not guessed any missed or correct letters yet. Then we call `getRandomWord(words)`, where `words` is a variable with the huge list of possible secret words we assigned on line 59. The return value of `getRandomWord(words)` is one of these words, and we save it to the `secretWord` variable. Then we also set a variable named `gameIsDone` to `False`. We will set `gameIsDone` to `True` when we want to signal that the game is over and the program should ask the player if they want to play again.

Setting the values of these variables is what we do before the player starts guessing letters.

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 we will always loop forever until a `break` statement is encountered. We will execute a `break` statement when the game is over (either because the player tbr-2 or they wayer tao(t)-2(a8-2()7(.J 12.12 -27.6 Td4 [(unt)L)1(s)-1j /2



Checking if the Letter is in the Secret Word

Because the player's guessed letter was

```
134. if len(missedLetters) == 6: # 6 is the last index in the
    HANGMANPICS list
```

But it is easier to just use `len(HANGMANPICS) - 1` instead.

So, when the length of the `missedLetters` string is equal to `len(HANGMANPICS) - 1`, we know the player has run out of guesses and has lost the game. We print a long

same word return from `getRandomWord()` twice in a row, but this is just a coincidence.

```
146.         else:
```


We have added two new multi

```
>>> stuff['hello']
```

```
>>>
```

As you can see, the expression `favorites1 == favorites2` evaluates to `True` because dictionaries are unordered, and they are considered to be the same if they have the



apples
42
cats

```
60. 'Shapes':'square triangle rectangle circle ellipse rhombus
    trapazoid chevron pentagon hexagon septagon octogon'.split
    (),
61. 'Fruits':'apple orange lemon lime pear watermelon grape
    grapefruit cherry banana cantalope 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()]
```

This code is put across multiple lines in the file, even though the Python interpreter thinks of it as just one "line of code." (The line of code doesn't end until the final } curly brace.)

The `random.choice()` Function

Now we will have to change our `getRandomWord()` function so that it chooses a random word from a dictionary of lists of strings, instead of from 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.
```



```

108.         correctLetters = ''
109.         secretWord = getRandomWord(words)
110.         gameIsDone = False

...

144.         gameIsDone = False
145.         secretWord = getRandomWord(words)
146.     else:

```

Because the `getRandomWord()` function now returns a list of two items instead of a string, `secretWord` will be assigned a list, not a string. We would then have to change the code as follows:

```

108. correctLetters = ''
109. secretWord = getRandomWord(words)

```




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

Add the line so your program looks like this:

```
122. while True:
123.     print('The secret word is in the set: ' + secretKey)
124.     displayBoard(HANGMANPICS, missedLetters,
                    correctLetters, secretWord)
```

Now we are done with our changes. Instead of just a single list of words, the secret word will be chosen from many different lists of words. We will also tell 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

Topics Covered In This Chapter:

Artificial Intelligence
Lis


```
0 | | 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 this source code and save it as *tictactoe.py*. Then run the game by pressing F5. You do not need to type in this program before reading this chapter. You can also download the source code by visiting the website at the URL <http://inventwithpython.com/chapter10> and following the instructions on the webpage.

tictactoe.py

This code can be downloaded from <http://inventwithpython.com/tictactoe.py>
If you get errors after typing this code in, compare it to the book's code with the online diff tool at <http://inventwithpython.com/diff> or email the author at al@inventwithpython.com

```
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.     # Let's 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 tuple 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

```
117.         if isWinner(copy, playerLetter):
118.             return i
119.
```

170.
1711

Computer's turn.

Tic Tac Toe is a very easy and short game to play on paper. In our Tic Tac Toe computer game, we'll let the player choose if they want to be X or O, randomly choose who goes first, and then let the player and computer take turns making moves on the board. Here is what a flow chart of this game could look like:

You can see a lot of the boxes on the left side of the chart are what happens during the player's turn. The right side of the chart shows what happens on the computer's turn. The player has an extra box for drawing the board because the computer doesn't need the board printed on the screen. After the player or computer makes a move, we check if they won or caused a tie, and then the game switches turns. If either the computer or player ties or wins the game, we ask the player if they want to play again.

Representing the Board as Data

First, we need to figure out how we are going to represent the board as 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 our program, we are going to represent the Tic Tac Toe board as a list of strings. Each string will represent one of the nine positions on the board. We will give a number to each of the spaces on the board. To make it easier to remember which index in the list is for

9. (Because there is no 0 on the keypad, we will just ignore the string at index 0 in our list.)

Game AI

When we talk about how our AI behaves, we will be talking about which types of spaces on the board it will move on. Just 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:

The AI for this game will follow a simple algorithm. An **algorithm** is a series of instructions to compute something. This is a very loose In the case of our Tic Tac Toe AI's algorithm, the series of steps will determine which is the best place to move. There is nothing in the code that says, "These lines are an algorithm." like there is with a function's def-block. We just consider the AI algorithm as all the code that is used in our program that determines the AI's next move.

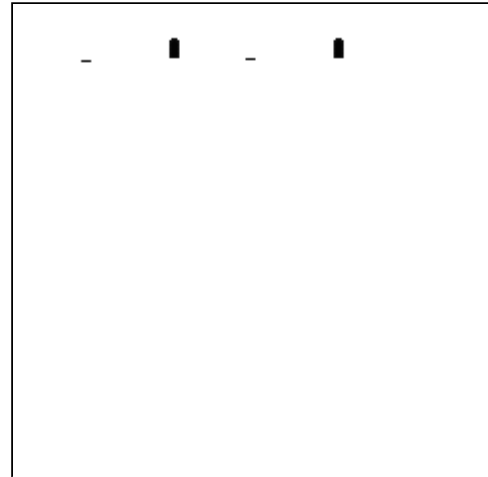


Figure 10-3: Locations of the side, corner, and center places.

Our algorithm will have the following steps:

1. First, see if there is 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 is a move the player can make that will cause the computer to lose the game. If there is, we should 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. (We always want to take a corner piece instead of the center or a side piece.) 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 we have reached step 5 the side spaces are the only spaces left.

This all takes place in the "Get computer's move." box on our flow chart. We could add this information to our flow chart like this:

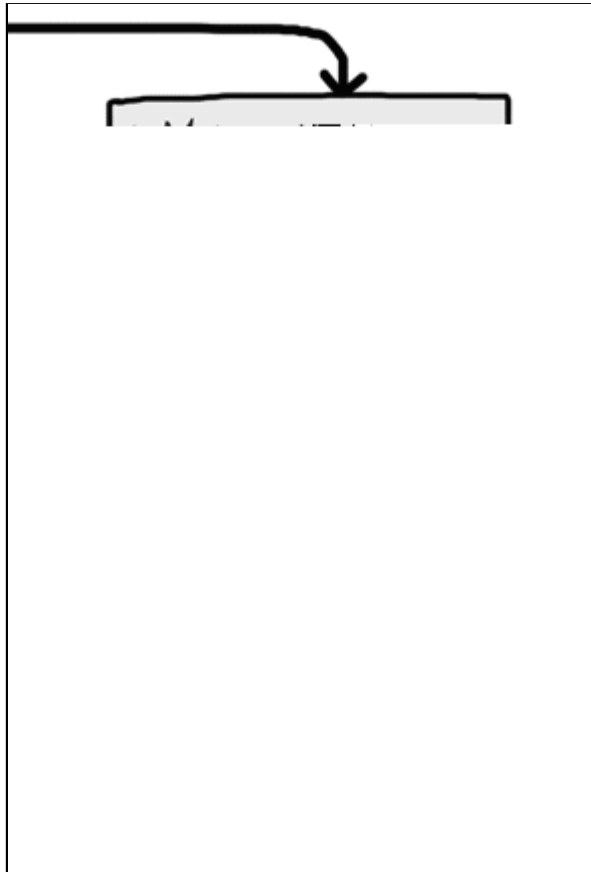


Figure 10-4: The five steps of the "Get computer's move" algorithm.
The arrows leaving go to the "Check if computer won" box.

We will implement this algorithm as code in our `getComputerMove()` function, and the other functions that `getComputerMove()` calls.

How the Code Works: Lines 1 to 81

Now that we know about how we want the program to work, let's look at what each line does.

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 we can use the `randint()` function in our game.

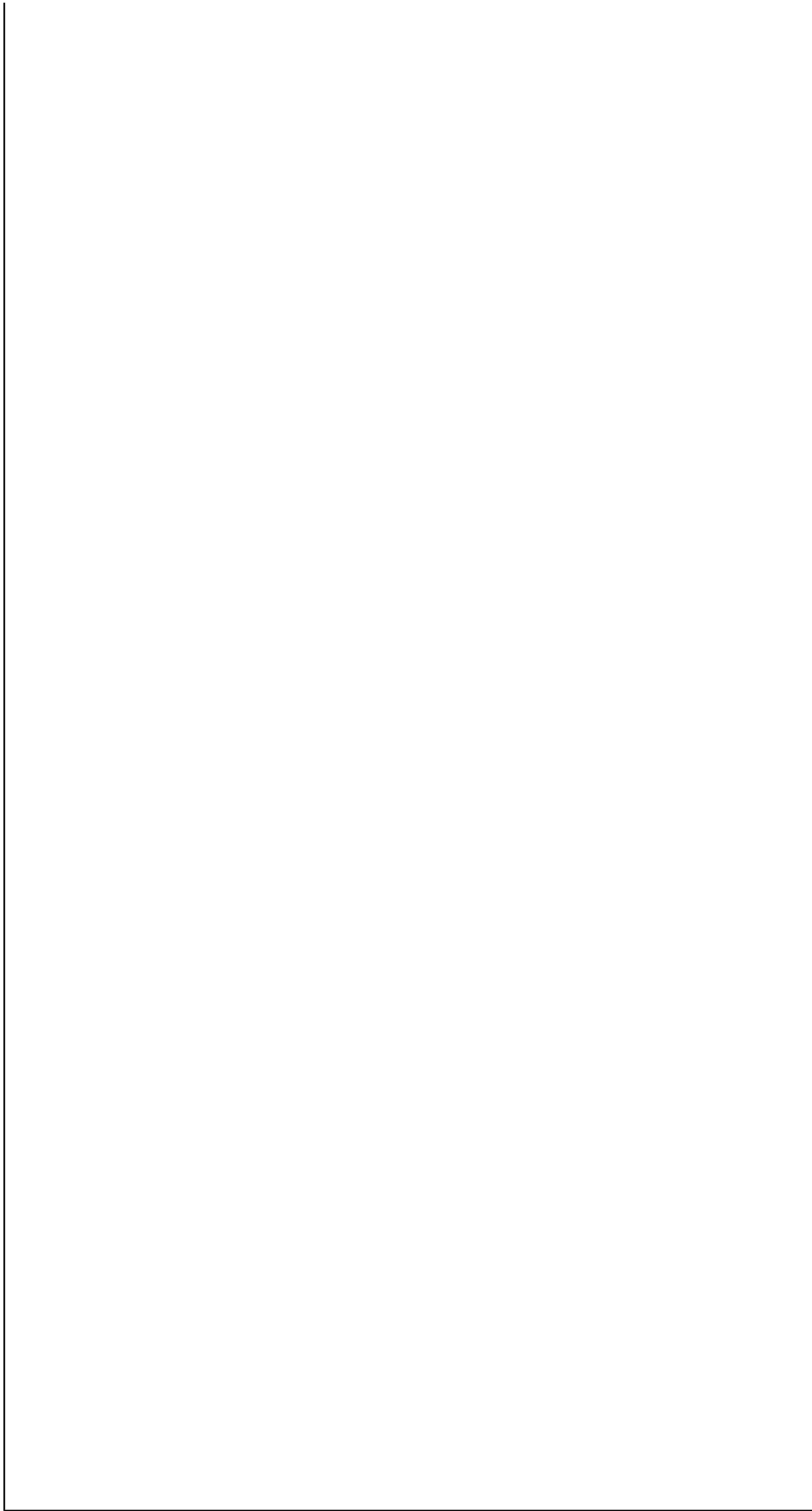
Printing the Board on the Screen

```
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('   |   |')
```

This function will print out the game board, marked as directed by the board parameter. Remember that our 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 remember that we ignore the string at index 0, because the spaces are labeled with numbers 1 to 9.) Many of our functions will work by passing the board as a list of ten strings to our functions. Be sure to get the spacing right in the strings that are printed, otherwise the board will look funny when it is printed on the screen.

Just as an example, here are some values that the board parameter could have (on the left) and what the drawBoard() function would print out:

at(r)34(s)-1(t7)3(4(u)-e)4(c)4(3(u)-e)(e)4(pp)g/T4_2 12 Tf o
drawBoard(board) cal



False and lets the program execution continue.







Figure 10-6: Two variables store two references to the same list.

When you assign the reference in `spam` to `cheese`, the `cheese` variable contains a copy of the reference in `spam`. Now both `cheese` and `spam` refer to the same list.



Figure 10-7: Changing the list changes all variables with references to that list.

When you alter the list that `cheese` refers to, the list that `spam` refers to is also changed because they are the same 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 will not 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, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 1, 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.

Using List References in `makeMove()`

Let's go back to the `makeMove()` function:

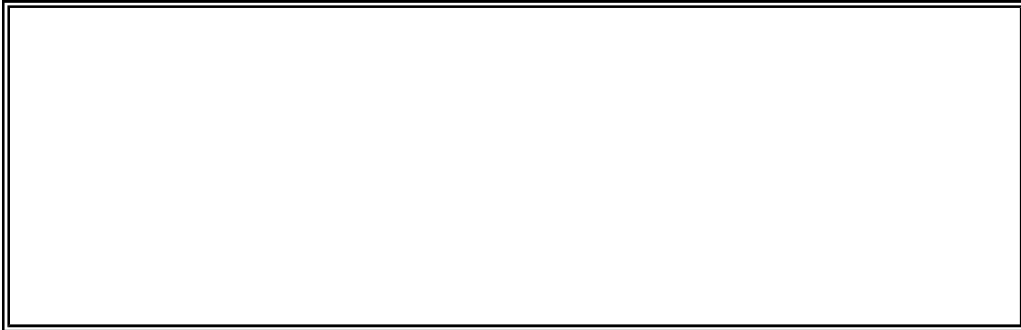
```
47. def makeMove(board, letter, move):  
48.     board[move] = letter
```

When we pass a list value as the argument for the `board` parameter, the function's local

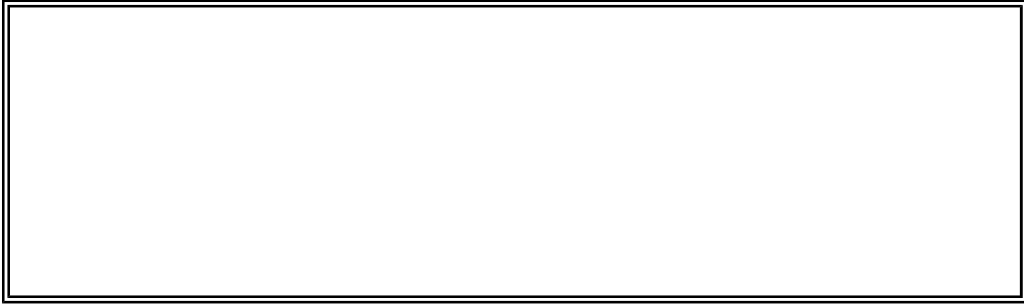
function. (This

Once again, we get rid of the parentheses, and we are left with one value:

```
53.     return True
```



Letting the Player Enter Their Move




```
'1 2 3 4 5 6 7 8 9'.split()
```


How the Code Works: Lines 83 to 94

The None Value

None is a special value that you can assign to a variable. The **None** value represents the lack of a value. None is the only value of the data type NoneType. (Just like the boolean data type has only two values, the NoneType data type has only one value, None.) It can be very useful to use the None

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 we have 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 immediately make that winning move. We will do this by trying each of the nine spaces on the board with a `for` loop. The first line in the loop makes a copy of the board list. We want to make a move on the copy of the board, and then see if that move results in the computer winning. We don't want to modify the original Tic Tac Toe board, which is why we make a call to `getBoardCopy()`. We check if the space we will move is free, and if so, we move on that space and see if this results in winning. If it does, we return that space's integer.

If moving on none of the spaces results in winning, then the loop will finally end and we move on to line 112.

The Computer Checks if the Player Can Win in One Move

```
112.     # Check if the player could win on his next move, and
113.     # block them.
114.     for i in range(1, 10):
115.         copy = getBoardCopy(board)
116.         if isSpaceFree(copy, i):
117.             makeMove(copy, playerLetter, i)
118.             if isWinner(copy, playerLetter):
119.                 return i
```

At this point, we know we cannot win in one move. So we want to make sure the human player cannot win in one more move. The code is very similar, except on the copy of the board, we place the player's letter before calling the `isWinner()` function. If there is a position the player can move that will let them win, the computer should move there to block that move.

If the human player cannot win in one more move, the for

index 0, which is just a placeholder that we ignore). If there is 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 spaces 1 through 9 on the Tic Tac Toe board. (Remember that range(1, 10) will make the for loop iterate over the integers 1, 2, 3, 4, 5, 6, 7, 8, and 9.) As soon as it finds a free space in 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, we will know that none of the spaces are free. So at that point, we will 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 is executed when we run this program.

```
142. while True:
143.     # Reset the board
144.     theBoard = [' '] * 10
```

This while loop has True for the condition, so that means we will keep looping in this loop until we encounter a break statement. Line 144 sets up the main Tic Tac Toe board that we will use, named theBoard. It is a 10-string list, where each string is a single space ' '. Remember the little trick using the multiplication operator with a list to replicate it: [' '] * 10. That evaluates to [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '], but is shorter for us to type [' '] * 10.

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']. We use


```
148.     gameIsPlaying = True
```



We want to check if the computer won with its last move. The reason we call `drawBoard()` here is because the player will want to see what move the computer made to win the game. We then set `gameIsPlaying` to `False` so that the game does not continue. Notice that lines 174 to 177 are almost identical to lines 157 to 160.

```
178.         else:
179.             if isBoardFull(theBoard):
180.                 drawBoard(theBoard)
181.                 print('The game is a tie!')
182.                 break
```

These lines of code are identical to the code on lines 162 to 165. The only difference is this is a check for a tied game after the computer has moved.

```
183.         else:
184.             turn = 'player'
```

If the game is neither won nor tied, it then becomes 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
```



```

    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

Here is a flow chart for this program. The flow chart describes the basic events of what happens in this game, and in what order they can happen:

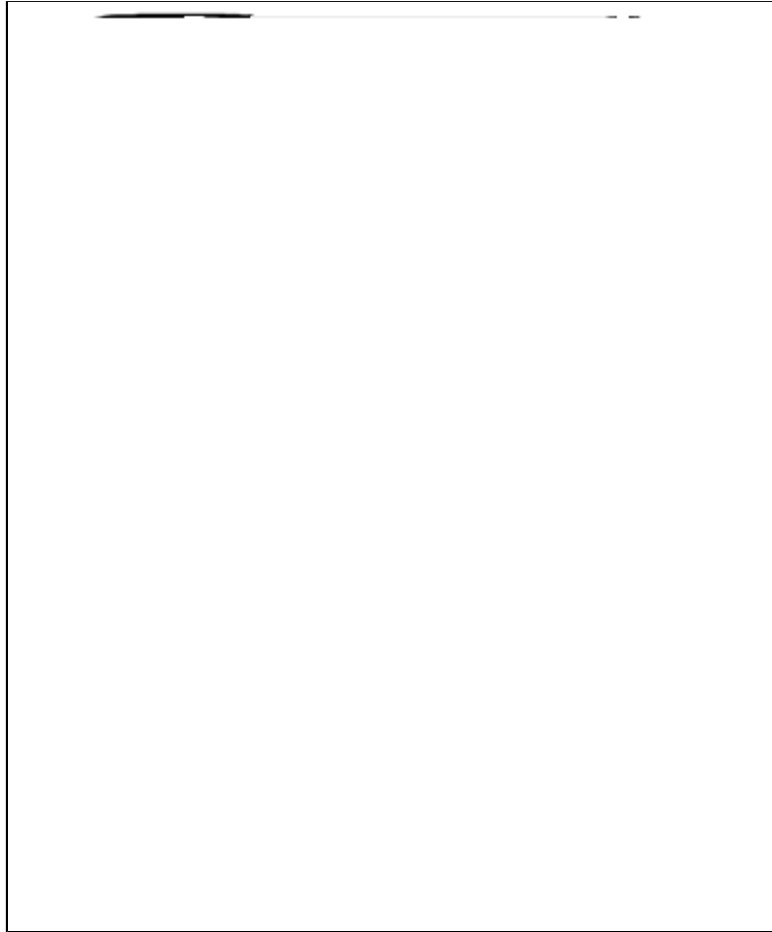


Figure 11-1: Flow chart for the Bagels game.

And here is the source code for our game. Start a new file and type the code in, and then save the file as *bagels.py*. We will design our game so that it is very easy to change the size of the secret number. It can be 3 digits or 5 digits or 30 digits. We will do this by using a constant variable named `NUMDIGITS` instead of hard-coding the integer 3 into our source code.

Hard-coding means writing a program in a way that it changing the behavior of the program requires changing a lot of the source code. For example, we could hard-code a name into a `print()` function call like: `print('Hello, Albert')`. Or we could use this line: `print('Hello, ' + name)` which would let us change the name that is printed by changing the name variable while the program is running.

How the Code Works: Lines 1 to 9

At the start of the program we import the `random` module and also create a function for generating a random secret number for the player to guess. The process of creating this number isn't hard, and also guarantees that it only has unique digits in it.


```
        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 what it is passed for the `guess` and `secretNum` parameters. The most obvious and easiest step is to check if the guess is the exact same as the secret number. In that case, we can just return `'You got it!'`.

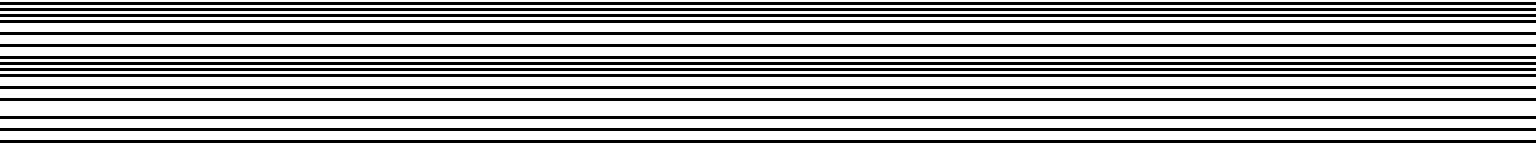
```
16.     clue = []  
17.  
18.     for i in range(len(guess)):  
19.         if guess[i] == secretNum[i]:  
20.             clue.append('Fermi')  
21.         elif guess[i] in secretNum:  
22.             clue.append('Pico')
```



```
>>> ' '.join(['My', 'name', 'is', 'Zophie'])
```

```
.....
```

```
.....
```



Finding out if the Player Wants to Play Again

```
40. def playAgain():
41.     # This function returns True if the player wants to play again
    return input("Do you want to play again? (y/n) ") == "y"
```

Q BTt(:).006.95


```
>>> name = 'Alice'
>>> event = 'party'
>>> where = 'the pool'
>>> when = 'Saturday'
>>> time = '6:00pm'
>>> print('Hello, ' + name + '. Will you go to the
' + event + ' at ' + where + ' this ' + when + '
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 very hard to type a line that concatenates several strings together. Instead, you can use **string interpolation**, which lets you put placeholders like `%s` (these placeholders are called **conversion specifiers**)

```
>>> 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
>>>
```

You get this error because string concatenation can only combine two strings, and >>>

our `getSecretNum()` function (passing it `NUMDIGITS` to tell how many digits we want the secret number to have) and assign it to `secretNum`. Remember that `secretNum` is a string, not an integer.

We tell the player how many digits is in our secret number by using string interpolation instead of string concatenation. We set a variable `numGuesses` to 1, to denote that this is the first guess. Then we enter a new `while` loop which will keep looping as long as `numGuesses` is less than or equal to `MAXGUESS`.

Getting the Player's Guess

Notice that this second `while` loop on line 60 is inside another `while` loop that started on line 55. Whenever we have these loops-inside-loops, we call them **nested loops**. You should know that any `break` or `continue` statements will only `break` or `continue` out of the innermost loop, and not any of the outer loops.

```
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. We will keep looping and asking the player for a guess until the player enters a guess that has the same number of digits as the secret number and is made up only of digits. This is what the `while` loop that starts on line 62 is for. We set `guess` as the blank string on line 61 so that the `while` loop's condition is `False` the first time, ensuring that we enter the loop at least once.

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 on line 62, we know that `guess` contains a valid guess. We pass this and the secret number in `secretNum` to our `getClues()` function. It returns a string that contains our clues, which we will display to the player. We


```
'helloXworldXyay'.
```

The `sort()` list method will rearrange the items in the list to be in alphabetical order.

The `append()` list method will add a value to the end of the associated list. If `spam` contains the list `['a', 'b', 'c']`, then calling `spam.append('d')` will change the list in `spam` to be `['a', 'b', 'c', 'd']`.

The next chapter is not 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 will be used in the Sonar, Reversi, and Dodger games, but Cartesian coordinates and negative numbers are used in almost all games (especially graphical games). If you already know about these concepts, give the next chapter a brief reading anyway just to freshen up. Let's dive in!

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 does not introduce a new game, but instead goes over some simple mathematical concepts that we will use in the rest of the games in this book.

Grids and Cartesian Coordinates

A problem in many games is how to talk about exact points on the board. A common way of solving this is by marking each individual row and column on a board with a letter and a number. Figure 12-1 is a chess board that has each row and each column marked.

In chess, the knight piece looks like a horse head. The white knight is located at the point e, 6 and the black knight is located at point a, 4. We can also see that every space on row 7 and every space in column c is empty.

A grid with labeled rows and columns like the chess board is a Cartesian coordinate system. By using a row label and column label, we can give a coordinate that is for one and only one space on the board. This can really help us describe to a computer the exact location we want. If you have learned about Cartesian coordinate systems in math class, you may know that usually we have numbers for both the rows and columns. This is handy, because otherwise after the 26th column we would run out of letters. That board would look like Figure 12-2.

The numbers going left and right that describe the columns are part of the **X-axis**. The numbers going up and down that describe the rows are part of the **Y-axis**. When we describe coordinates, we always say the X-coordinate first, followed by the Y-coordinate. That means the white knight in the above picture is located at the coordinate 5, 6 (and not 6, 5). The black knight is located at the coordinate 1, 4 (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 then to the right by four spaces. (Or move right four spaces and then move up two spaces.) But we don't need to look at the board to figure this out. If we know the white knight is located at 5, 6 and the black knight is located at 1, 4, then we can just use subtraction to figure out this information.

Subtract the black knight's X-coordinate and white knight's X-coordinate: $5 - 1 = 4$. That

Negative Numbers

Another concept that Cartesian coordinates use is negative numbers.

Negative numbers are numbers that are smaller than zero. We put a minus sign in front of a number to show that it is a negative number. -1 is smaller than 0. And -2 is smaller than -1. And -3 is smaller than -2. 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 is not positive or negative. In this picture, you can see the positive numbers increasing to the right and the negative numbers decreasing to the left:

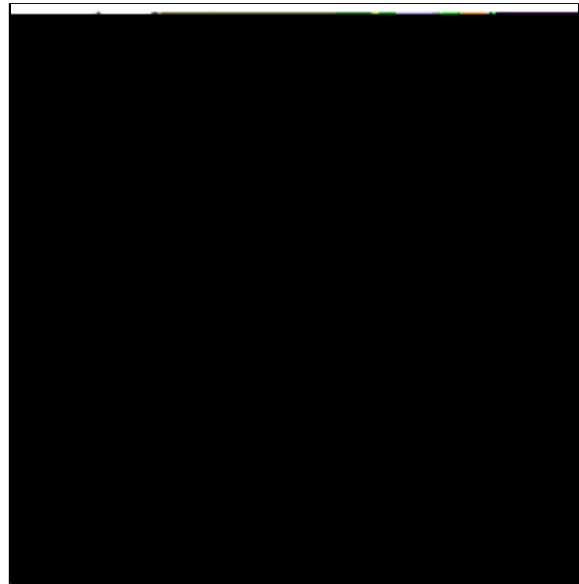


Figure 12-2: The same chessboard but with numeric coordinates for both rows and columns.

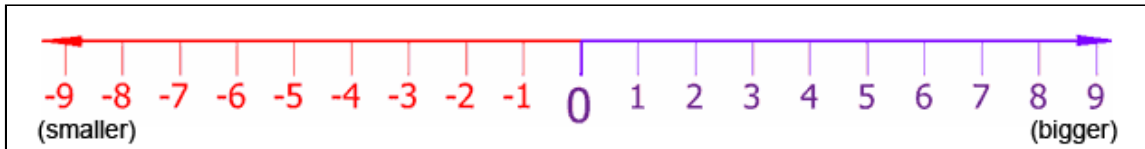


Figure 12-3: A number line.

The number line is really useful for doing subtraction and addition 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).

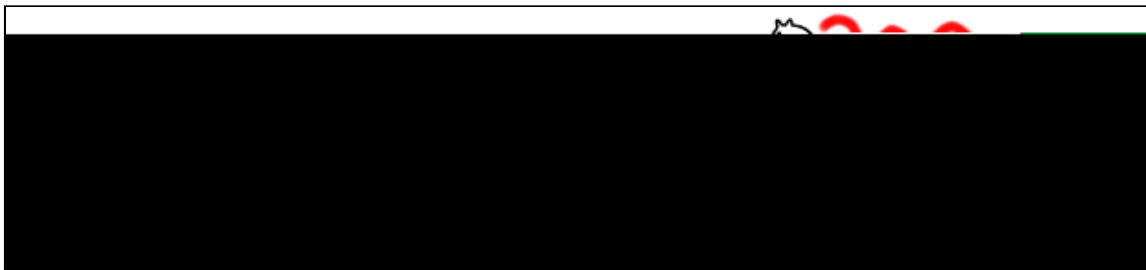


Figure 12-4: Moving the white knight to the right adds to the coordinate.

As you can see, the white knight ends up at position 7. This makes sense, because $4 + 3$ is 7.

Subtraction can be 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:

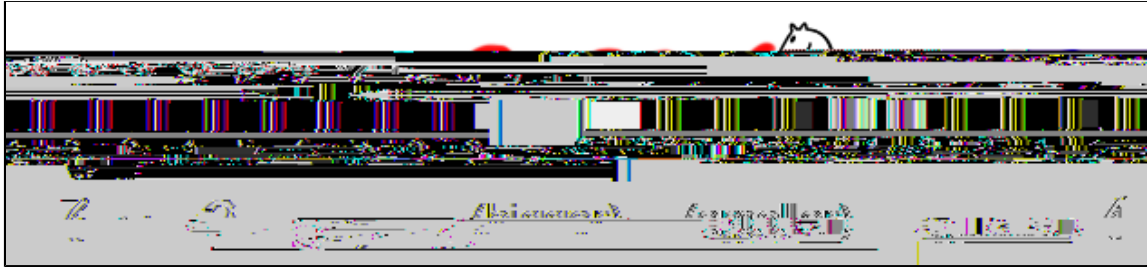


Figure 12-5: Moving the white knight to the left subtracts from the coordinate.

The white knight ends up at position -2 . That means $4 - 6$ equals -2 .

If we 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.

The number line is the same as the X-axis. If we made the number line go up and down

instead of left and right, it would model the Y-axis. Adding a positive number (or

Figure 12-9: Trick 2 - Subtracting a positive and negative number.

Trick 3: The Commutative Property of Addition

A third trick is to remember that when you add two numbers like 6 and 4, it doesn't matter what order they are in. (This is called the **commutative property** of addition.) That means that $6 + 4$ and $4 + 6$ both equal the same value, 10. If you count the boxes in the figure below, you can see that it doesn't matter what order you have the numbers for addition.

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. -

Figure 12-11: Using our math tricks together.

black knight is at position -

Summary: Using this Math in Games

This hasn't been too much math to learn for programming. In fact, most programming does not

Topics Covered In This Chapter:

Data structures.

The `remove()` list method.

The `isdigit()` string method.

The `sys.exit()` function.

The game in this chapter only introduces a couple new helpful methods that come with Python, the `remove()` list method and the `isdigit()` string method. But this is the

Figure 13-1: The first sonar device shows a ring of possible places the treasure could be located.

Figure 13-2: Combining the rings of all three sonar devices shows only one possible place for the treasure.


```
27.     # print the numbers across the bottom
28.     print()
```

```

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] = '0'
91.             return 'Sonar did not detect anything. All
treasure chests out of range.'
92.
93.
94. def enterPlayerMove():
95.     # Let the player type in her 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 = inu2Yih]82 82de
99.
96 he s cf((-2(0)-2(-)-2(5)    # derellesttD-2(y)-2(l))-14( )-2(=)-2( )-2(')-2(0)-2
device? (0-59 0-1. dere

device? ( inu2Yih]82) ' s14( )-2(=)-2( -2(e)-(r)-i2(e)-2)-2( )-2( )]TJ -260

```

```

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.     4444444444
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.     4444444444
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.

```



```

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

Sonar is kind of complicated, so it might be better to type in the game's code and play it a few times first to understand what is going on. After you've played the game a few times, you can kind of get an idea of the sequence of events in this game.

The Sonar game uses lists of lists and other complicated variables. These complicated variables are known as **data structures**. Data structures will let us store nontrivial arrangements of values in a single variable. We will use data structures for the Sonar board and the locations of the treasure chests. One example of a data structure was the board variable in the Tic Tac Toe chapter.

It is also helpful to write out the things we need our program to do, and come up with some function names that will handle these actions. Remember to name functions after what they specifically do. Otherwise we might end up forgetting a function, or typing in two different functions that do the same thing.

Table 13-1: A list of each function the Sonar game needs.

What the code should do.	The function that will do it.
Prints the game board on the screen based on the board data structure it is passed, including the coordinates along the top, bottom, and left and right sides.	<code>drawBoard()</code>
Create a fresh board data structure.	<code>getNewBoard()</code>
Create a fresh <code>chests</code> data structure that has a number of chests randomly scattered across the game board.	<code>getRandomChests()</code>
Check that the XY coordinates that are passed to this function are located on the game board or not.	<code>isValidMove()</code>
Let the player type in the XY coordinates of his next	

move, and keep asking until they type in the coordinates correctly.	<code>enterPlayerMove()</code>
Place a sonar device on the game board, and update the board data structure then return a string that describes what happened.	<code>makeMove()</code>
Ask the player if they want to play another game of Sonar.	<code>playAgain()</code>
Print out instructions for the game.	<code>showInstructions()</code>

These might not be all of the functions we need, but a list like this is a good idea to help you get started with programming your own games. For example, when we are writing the `drawBoard()` function in the Sonar game, we figure out that we also need a `getRow()` function. Writing out a function once and then calling it twice is preferable to writing out the code twice. The whole point of functions is to reduce duplicate code down to one place, so if we ever need to make changes to that code we only need to change one place in our program.

How the Code Works: Lines 1 to 38

```
1. # Sonar
2.
3. import random
4. import sys
```

Here we import two modules, `random` and `sys`. The `sys` module contains the `exit()` function, which causes the program to immediately terminate. We will call this function later in our program.

Drawing the Game Board

```
6. def drawBoard(board):
```

The back tick (```) and tilde (`~`) characters are located next to the 1 key on your keyboard. They resemble the waves of the ocean. Somewhere in this ocean are three treasure chests, but you don't know where. You can figure it out by planting sonar devices, and tell the game program where by typing in the X and Y coordinates (which are printed on the four sides of the screen.)

The `drawBoard()` function is the first function we will define for our program. The sonar game's board is an ASCII-art ocean with coordinates going along the X- and Y-axis, and looks like this:

W(he)4(e)-6(s)2TJ -2Tles eisheesvi28(i)-6dsW1sD1s

The numbers on the first line which mark the tens position all have nine spaces in between them, and there are thirteen spaces in front of the 1. We are going to create a string with thi

we assign a single space string to `extraSpace`. Otherwise, we set `extraSpace` to be a blank string. This way, all of our rows will line up when we print them.

The `getRow()` function will return a string representing the row number we pass it. Its two parameters are the board data structure stored in the `board` variable and a row

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([])
```

At the start of each new game, we will need a fresh board data structure. The board data structure is a list of lists of strings. The first list represents the X coordinate. Since our game's board is 60 characters across, this first list needs to contain 60 lists. So we 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 the Y coordinate of our game board. There are 15 rows in the board, so each of these 60 lists must have 15 characters in them. We have another for loop to add 15 single-character strings that represent the ocean. The "ocean" will just be a bunch of '~' and '`' strings, so we will randomly choose between those two. We can do this by using the random module.

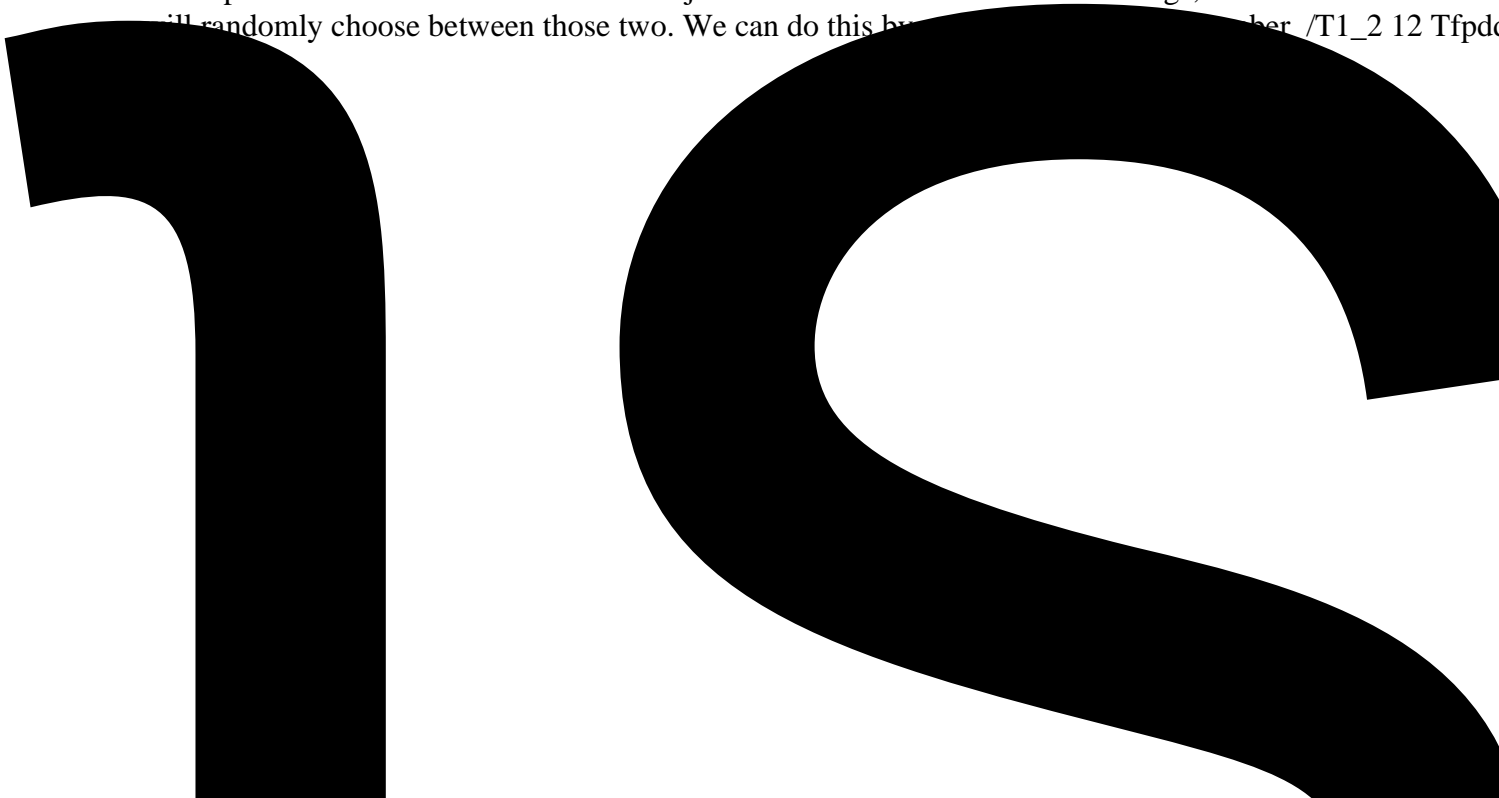


Figure 13-4: The indexes of a list of lists.

51.	return board
-----	--------------

We will pass the `numChests` parameter to tell the function how many treasure chests we want it to generate. We set up a `for` loop to iterate this number of times, and on each iteration we append 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 something like `[2, 2]` or `[2, 4]` or `[10, 0]`. This data structure is then returned.

`makeMove()` will return `False`.

If the coordinates land directly on the treasure, `makeMove()` will return the string 'You have found a sunken treasure chest!'. If the XY coordinates are within a distance of 9 or less of a treasure chest, we return the string 'Treasure detected at a distance of %s from the sonar device.' (where %s is the distance). Otherwise, `makeMove()` will return the string '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 XY coordinates of where the player wants to drop the sonar device, and a list of XY coordinates for the treasure chests (in the `chests` list of lists), how do we find out which treasure chest is closest?

An Algorithm for Finding the Closest Treasure Chest

While the `x` and `y` variables are just integers (say, 5 and 0), together they represent the location on the game board (which is a Cartesian coordinate system) where the player guessed. The `chests` variable may have a value such as `[[5, 0], [0, 2], [4, 2]]`, that value represents the locations of three treasure chests. Even though these variables are a bunch of numbers, we can visualize it like this:

	0	1	2	3	4	5
0	2	2	2	3	4	5,2
1	1	1	2	3	4	5
2	0,0	1	2	3	4,0	5
3	1	1	2	3	4	5
4	2	2	2	3	4	5
5	3	3	3	3	4	5

Figure 13-5: The places on the board that $[[5, 0], [0, 2], [4, 2]]$ represents.

We figure out the distance from the sonar device located at 0, 2 with "rings" and the distances around it:

	0	1	2	3	4	5

Figure 13-6: The board marked with distances from the 0, 2 position.

But how do we translate this into code for our game? We need a way to represent 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 we should subtract the sonar device's X coordinate and a treasure chest's X coordinate, and then take the absolute value of this number. We 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. Let's look at our example board with rings above to see if this algorithm is correct.

The sonar's X and Y coordinates are 3 and 2. The first treasure chest's X and Y coordinates (first in the list `[[5, 0], [0, 2], [4, 2]]` that is) are 5 and 0.

For the X coordinates, $3 - 5$ evaluates to -2 , and the absolute value of -2 is 2.

For the Y coordinates, $2 - 1$ evaluates to 1, and the absolute value of 1 is 1.

Comparing the two absolute values 2 and 1, 2 is the larger value and should be the distance from the sonar device and the treasure chest at coordinates 5, 1. We can look at the board 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 their distances work out correctly also.

Let's find the distance from the sonar device at 3,2 and the treasure chest at 0,2. `abs(3 - 0)` evaluates to 3. The `abs()` function returns the absolute value of the number we pass to it. `abs(2 - 2)` evaluates to 0. 3 is larger than 0, so the distance from the sonar

The for loop `for cx, cy in chests:` combines both of these principles. 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]]`, on the first iteration through the loop, `cx` will have the value 5 and `cy` will have the value 0.

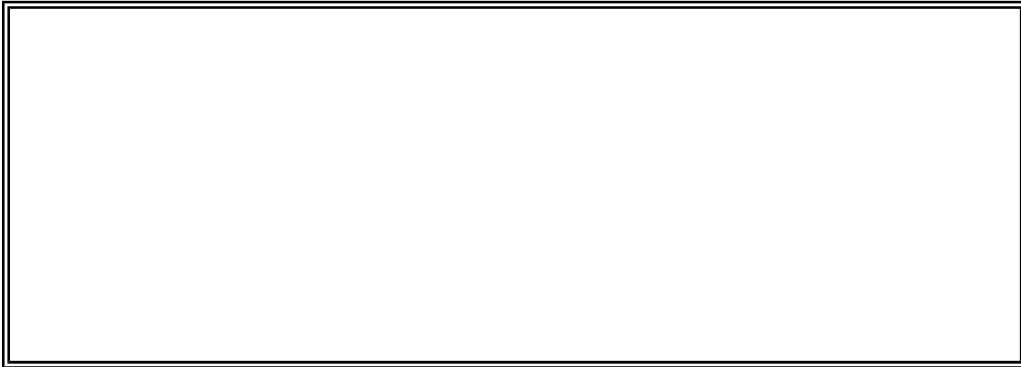
Line 73 determines which is larger: the absolute value of the difference of the X coordinates, or the absolute value of the difference of the Y coordinates. (if

You can see that 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 shell:

How the Code Works: Lines 94 to 162

The last few functions we need are to let the player enter their move on the game board, ask the player if he wants to play again (this will be called at the end of the game), and print the instructions for the game on the screen (this will be called at the beginning of the game).

Getting the Player's Move




```
        locate the next
159. closest sunken treasure chest.
160. Press enter to continue...'''
161.     input()
162.     print()
```

This is the rest of the instructions in one multi-line string. After the player presses Enter, the function returns. These are all of the functions we will define for our game. The rest of the program is the main part of our game.

How the Code Works: Lines 165 to 217

Now that we are done writing all of the functions our game will need, let's start the main part of the program.

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()
```


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))
```

This while loop executes as long as the player has sonar devices remaining. We want to print a message telling the user how many sonar devices and treasure chests are left. But there is a problem. If there are two or more sonar devices left, we want to print '2 sonar

devices'. Be(e)4TJ 12 Tf [(B)'2 sonar e.((e)4TJ 12T1_2 12 T6o1(a)-6(pr)3(()fI0(g)40g)0(p)81


```
204.         print('You have found all the sunken treasure  
           chests! Congratulations and good game!')  
205.         break
```



The second place is the `break` statement on line 205. That statement is executed if the player has found all the treasure chests before running out of sonar devices. In that case, `sonarDevices` would be some value greater than 0.

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 know where the treasure chests had been lurking.

Asking the Player to Play Again, and the `sys.exit()` Function

```
216.     if not playAgain():
217.         sys.exit()
```

Win or lose, we call the `playAgain()` function to let the player type in whether they want to keep playing or not. If not, then `playAgain()` returns `False`. The not operator 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.

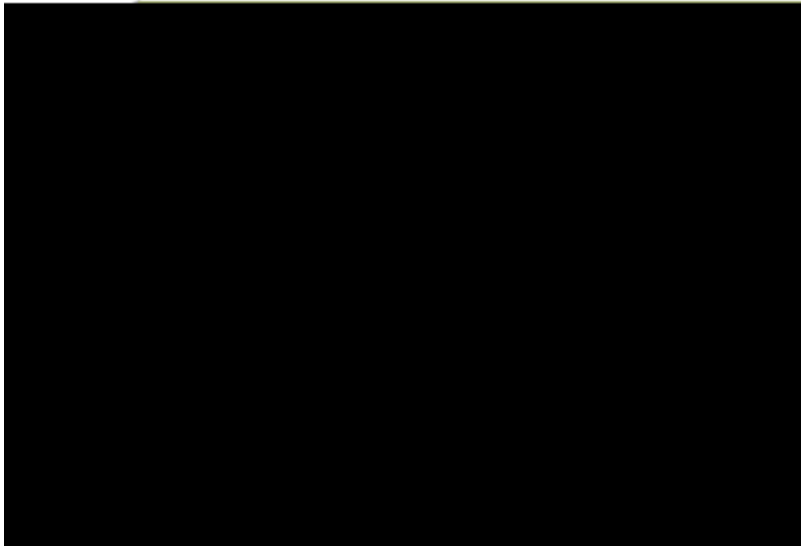
Summary: Review of our Sonar Game

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 nine hundred 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.

Game boards in games that use a Cartesian coordinate system are often 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 in our program, and our 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 mathematically operations on them which will encrypt or decrypt secret messages.



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 is not really a game, but it is fun to play with nonetheless. Our program will convert normal English into a secret code, and 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 in order to convert it into secret messages, we will learn several new functions and methods that come with Python for manipulating strings. We will also learn how programs can do math with text strings just as it can with numbers.

About Cryptography

The science of writing secret codes is called **cryptography**. Cryptography has been

used for thousands of years to send secret messages that only the recipient could understand, even if someone captured the messenger and read the coded message. A secret code system is called a **cipher**. There are thousands of different ciphers that have been used, each using different techniques to keep the messages a secret.

In cryptography, we call the message that we want to be secret the **plaintext**. The plaintext could look something like this:

```
Hello there! The keys to the house are hidden under the reddish flower pot.
```

When we convert the plaintext into the encoded message, we call this **encrypting** the plaintext. The plaintext is encrypted into the **ciphertext**. The ciphertext looks like random letters (also called **garbage data**), and we cannot understand what the original plaintext was by just looking at the ciphertext. Here is an example of some ciphertext:

```
Ckkz f kx kj becqnejc kqp pdeo oaynap iaowca!
```

But if we know about the cipher used to encrypt the message, we 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. Although all the door locks of the same type are built the same, but a particular lock will only unlock if you have the key made for that lock.

The Caesar Cipher

When we encrypt a message using a cipher, we will choose the key that is used to encrypt and decrypt this message. The key for our Caesar Cipher will be a number from 1 to 26. Unless you know the key (that is, know the number), you will not be able to decrypt the encrypted message.

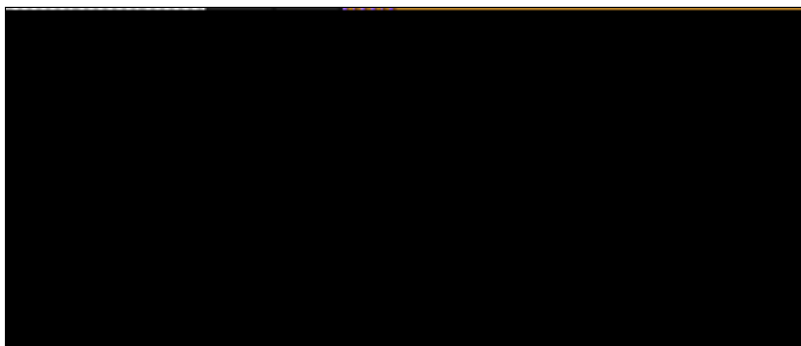


Figure 14-1: Shifting over letters by three spaces. Here, B becomes E.

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 3 spaces.

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 of spaces over. When you get to the leftover letters at the end, wrap around back to the start of the boxes. Here is an example with the letters shifted by three spaces:

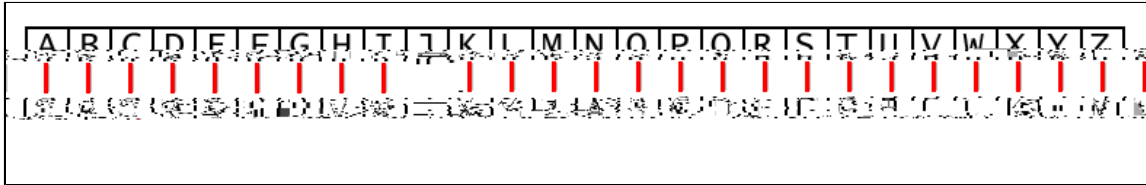


Figure 14-2: The entire alphabet shifted by three spaces.

The number of spaces we shift is the key in the Caesar Cipher. The example above shows the letter translations for the key 3.

Using a key of 3, if we encrypt the plaintext "Howdy", then the "H" becomes "E". The letter "o" becomes "l". The letter "w" becomes "t". The letter "d" becomes "a". And the letter "y" becomes "v". The ciphertext of "Hello" with key 3 becomes "Eltav".

We will keep any non-letter characters the same. In order to decrypt "Eltav" with the key 3, we just go from the bottom boxes back to the top. The letter "E" becomes "H", the letter "l" becomes "o", the letter "t" becomes "w", the letter "a" becomes "d", and the letter "v" becomes "y" to form "Howdy".

You can find out more about the Caesar Cipher from Wikipedia at http://en.wikipedia.org/wiki/Caesar_cipher

ASCII, and Using Numbers for Letters

How do we implement this shifting of the letters in our program? 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 number (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 127. The numbers less than 32 refer to "unprintable" characters, so we will not be using them.

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-

So if we wanted to shift "A" by three spaces, we first convert it to a number (65). Then we add 3 to 65, to get 68. Then we convert the number 68 back to a letter ("D"). We will use the `chr()` and `ord()` functions to convert between letters and numbers.

For example, the letter "A" is represented by the number 65. The letter "m" is represented by the number 109. A table of all the ASCII characters from 32 to 127 is in Table 14-1.

The `chr()` and `ord()` Functions

The `chr()` function (pronounced "char", short for "character") takes an integer ASCII number for the parameter and returns the single-character string. The `ord()` function (short for "ordinal") takes a single-character string for the parameter, and returns the integer ASCII value for that character. Try typing 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
>>>
```

key. Remember that if you do not know the correct key, the decrypted text will just be garbage data.

Do you wish to encrypt or decrypt a message?

decrypt

```

    (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: Lines 1 to 34

This code is much shorter compared to our other games. The encryption and decryption processes are the just 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 simply a comment. The Caesar Cipher is one cipher of a type of ciphers called simple substitution ciphers. **Simple substitution ciphers** are ciphers that replace

Getting the Key from the Player

```
18. def getKey():
19.     key = 0
20.     while True:
21.         print('Enter the key number (1-%s)' %
22.             (MAX_KEY_SIZE))
23.         key = int(input())
24.         if (key >= 1 and key <= MAX_KEY_SIZE):
25.             return key
```

The `getKey()` function lets the player type in key they will use to encrypt or decrypt the message. The `while` loop ensures that the function only returns 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 have the value 26 because it is constant). It then returns this key.

the value of `num` here would be the character `'^'` (The ordinal of `'^'` is 94). But `^` isn't a letter at all. We wanted the ciphertext to "wrap around" to the beginning of the alphabet.

The way we can do this is to check if `key` has a value larger than the largest possible


```
55.  
56. print('Your translated text is:')  
57. print(getTranslatedMessage(mode, message, key))
```

This is the main part of our program. We call each of the three functions we have defined above in turn to get the mode, message, and key that the user wants to use. We then pass these three values as arguments 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:
```

```

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 will let us select "brute force" as a mode for our program. Then 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 make our program ask the user for a key if they are not in "brute force" mode. If they are not in "brute force" mode, then the original `getTranslatedMessage ()` call is made and the translated string is printed.

However, otherwise we are in "brute force" mode, and we run a `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 we have `+ 1`. This program will print out 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 xtmiavb, jcb kmzbiqvbq qa ijaczl.

Your translated text is:

1 Kvbiaz thf uva il wslhzhua, iba jlyahpuaf pz hizbyk.

But while our Caesar

Topics Covered In This Chapter:

The `bool()` Function

Evaluating Non-Boolean Values as Booleans

How to Play Reversi

In this chapter we will make a game called Reversi. Reversi (also called Othello) is a



Figure 15-1: The starting Reversi board has two white tiles and two black tiles.

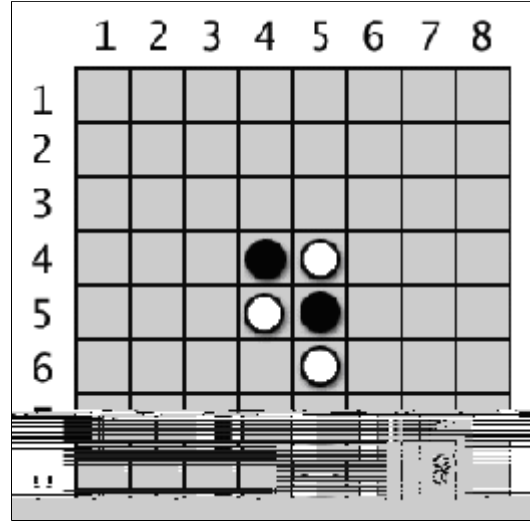


Figure 15-2: White places a new tile.

The black tile at 5, 5 is in 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 in between the player's new tile and existing tile. Below 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.

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

X scored 15 points. O scored 46 points.
You lost. The computer beat you by 31 points.


```

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.    # Let's the player type which tile they want to be.

```

```
131.     # Returns a list with the player's tile as the first
132.     item, and the computer's tile as the second.
133.     tile = ''
134.     while not (tile == 'X' or tile == 'O'):
135.         print('Do you want to be X or O?')
136.         tile = input().upper()
137.     # the first element in the tuple is the player's tile,
138.     the second is the computer's tile.
139.     if tile == 'X':
140.         return ['X', 'O']
141.     else:
142.         return ['O', 'X']
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
154.     play again, otherwise it returns False.
155.     print('Do you want to play again? (yes or no)')
156.     return input().lower().startswith('y')
157.
158. def makeMove(board, tile, xstart, ystart):
159.     # Place the tile on the board at xstart, ystart, and
160.     flip any of the opponent's pieces.
161.     # Returns False if this is an invalid move, True if it
162.     is valid.
163.     tilesToFlip = isValidMove(board, tile, xstart, ystart)
164.     if tilesToFlip == False:
165.         return False
```

```
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':
```

```
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:
```



```

    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 (p)-2(r)-2(i)-2(n)-2(t)-2(()-2(')-2('getScoreOfBoa
293.
2rd(27dtHs-2(1o)-2(r)- 2(s)-2(l)-2(plC(r)-2(T)-2(i)-t2(o) (a)-2(k)-1(%)-2( )]TJ
292( )-2( )-2( )-2( )-2( )-2( )-2(e)-2(l)-2(T)-22e
295.  louYorer (r)-2(: )-2(s)-2(c)-2(o)-2(r)-2(e)T

```



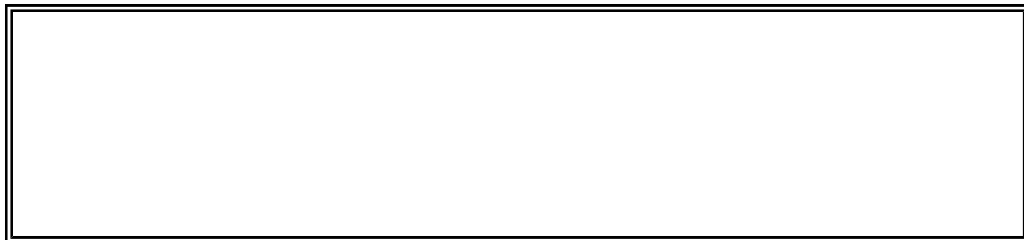
```
14.         print(VLINE)
15.         print(y+1, end=' ')
16.         for x in range(8):
17.             print(' | %s' % (board[x][y]), end=' ')
18.         print(' | ')
19.         print(VLINE)
20.         print(HLINE)
```

Printing each row of spaces on the board is fairly repetitive, so we can use a loop here. We 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 a comma at the end of it to prevent a new line. This is so we can



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

Resettinl44 -608 I -3156 -600 I 0 -600 I f* Q BT /T1_0 9.96 Tf 104.28 415.08 Td [(+)-



When we start a new game of Reversi, it isn't enough to have a completely blank board.

structure, the player's tile, and the XY coordinates for player's move, this function should return `True` if the Reversi game rules allow that move and `False` if they don't.

The easiest check we can do to disqualify a move is to see if the XY coordinates are on the game board or if the space at XY is not empty. This is what the `if` statement on line 48 checks for. `isOnBoard()` is a function we will write that makes sure both the X and Y coordinates are between 0 and 7.

For the purposes of this function, we will go ahead and mark the XY coordinate pointed to by `xstart` and `ystart` with the player's tile. We set this place on the board back to a space before we leave this function.

The player's tile has been passed to us, but we will need to be able to identify the other player's tile. If the player's tile is 'X' then obviously the other player's tile is 'O'. And it is the same the other way.

Finally, if the given XY coordinate ends up as a valid position, we will return 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. We will move around the board in a direction by adding the first value in the two--6(ga7t)-2(e)4(m)-4(l)-2(i)-2(s)-1(t)-0(t)8-7(oy a)-6 h ae 8ot-2(e-c)-2(t) y a5 othert vaspav8(

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. We will change `x` and `y` to "move" in the direction that `xdirection` and `ydirection` dictate. `xstart` and `ystart` will stay the same so we can remember which space we originally intended to check. (Remember, we need to set this place back to a space character, so we shouldn't overwrite the values in them.)

But if the first space does have the other player's tile, then we should keep proceeding in that direction until we reach one of our own tiles. If we move off of the board, then we 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 ensures that `x` and `y` keep going in the current direction as long as we keep seeing a trail of the other player's tiles. If `x` and `y` move off of the board, we break out of the `for` loop and the flow of execution moves to line 74. What we really want to do is break out of the `while` loop but continue in the `for` loop. But if we put a `continue` statement on line 73, that would only continue to the `while` loop on line 69.

Instead, we recheck `not isOnBoard(x, y)` on line 74 and then continue from there, which goes to the next direction in the `for` statement. It is important to know that `break` and `continue` will only break or continue in the loop they are called from, and not an outer loop that contains the loop they are called from.

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])
```

If the `while` loop on line 69 stopped looping because the condition was `False`, then we have found a space on the board that holds our own tile or a blank space. Line 76 checks if this space on the board holds one of our tiles. If it does, then we have found a valid move. We start a new `while` loop, this time subtracting `x` and `y` to move them in the opposite direction they were originally going. We note each space between our tiles on the board by appending the space to the `tilesToFlip` list.

We break out of the `while` loop once `x` and `y` have returned to the original position (which was still stored in `xstart` and `ystart`).


```

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

```

We perform this check in all eight directions, and afterwards 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 does not actually 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` would be an empty list and this move would not be valid. In that case, `isValidMove()` should return `False`. Otherwise, we should return `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()`, and is just shorthand for the rather complicated Boolean expression that returns `True` if both `x` and `y` are in between 0 and 7. This function lets us make sure that the coordinates are actually on the game board.

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

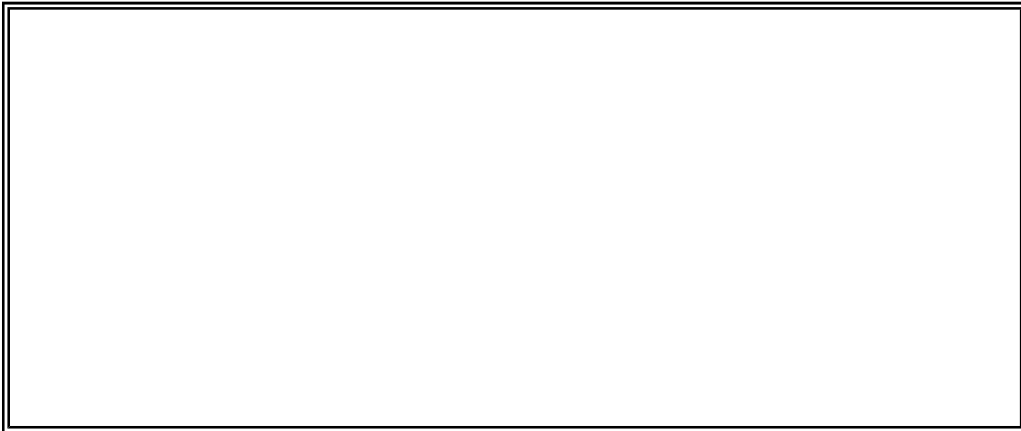
```

`getBoardWithValidMoves()` is used to return a game board d 7

modifying the one passed to it by the board parameter. Line 100 calls `getValidMoves()`, which returns a list of xy coordinates with all the legal moves the playe

```
False
>>> bool({})
False
>>> bool(1)
True
>>> bool('Hello')
True
>>> bool([1, 2, 3, 4, 5])
True
>>> bool({'spam': 'cheese', 'fizz': 'buzz'})
True
>>>
```

Whenever you have a condition, imagine that the entire condition is placed inside a call to `bool()` as the parameter. Conditions are automatically interpreted as Boolean values. This is similar to how `print()`



might be a bit confusing,

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')

```

We have used the `playAgain()` in our previous games. If the player types in something 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 the function we call when we 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 as a parameter directly. Changes made to the board variable (because it is a list) will be made to the global scope as well. 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

```

If the return value of `isValidMove()` was `False`, then `makeMove()` will also return `False`.

Otherwise, `isValidMove()` would have returned a list of spaces on the board to put down our tiles (the 'X' or 'O' string in `tile`). Line 166 sets the space that the player has moved on, and the `for` loop after that sets all the tiles that are in `tilesToFlip`.

Copying the

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']`. We create this constant because it is easier type `DIGITS1TO8` than the entire list.

```
192.      whi(a)4(n a1199951 0 0 0.119y1 -8 o -8u9951 0 0 :) -2(w) -2(h)28 TD [(2u
```

numbers beginning at 1 instead of 0. So when we convert the strings in

possible moves that the computer can make. We will select the best move from this list.

```
219.     # randomize the order of the possible moves
220.     random.shuffle(possibleMoves)
```

First, we a14()*do g o u se t

If there are no corner moves, we will go through the entire list and find out which move gives us the highest score. The `for` loop will set `x` and `y` to every move in `possibleMoves`. `bestMove` will be set to the highest scoring move we've found so far, and `bestScore` will be set to the best move's score. When the code in the loop finds a move that scores higher than `bestScore`

We do not always want to go with the first move in the `possibleMoves` list, because that would make our AI predictable by the player. But it is random, because on line 220 we shuffled the `possibleMoves` list. Even though our code always chooses the first of these tied moves, it is random which of the moves will be first in the list because the order is random. This ensures that the AI will not be predictable when there is 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()` simply calls the `getScoreOfBoard()` function and then prints out the player's score and the computer's score. 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 we define for our Reversi game. The code starting on line 246 will implement the actual game and make calls to these functions when they are 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

The turn variable is a string will either have the string value 'player' or 'computer', and will keep track of whose turn it is. We set turn to the return value of `whoGoesFirst()`, which randomly chooses who will go first. We then print out who goes first to the player on line 255.

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. We 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.) The first thing we want to do is display the board to the player. If hints mode is on (which it is if `showHints` is `True`), then we want to get a board data structure that has '.' period characters on every space the player could go.

Our `getBoardWithValidMoves()` function does that, all we have to do is pass the game board data structure and it will return a copy that also contains '.' period characters. We then pass this board to the `drawBoard()` function.

If hints mode is off, then we just pass `mainBoard` to `drawBoard()`.

After printing out the game board to the player, we also want to print out the current

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, we should call 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, we want to turn hints mode on (if it was off) or off (if it was on). The `showHints = not showHints` assignment statement handles both of these cases, because `not False` evaluates to `True` and `not True` evaluates to `False`. Then we run the `continue` statement to loop back (turn has not changed, so it will still be the player's turn after we loop).

Make the Player's Move

Otherwise, if the player did not quit or toggle hints mode, then we will call `makeMove()` to make t

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)
```

The first thing we do when it is the computer's turn is call `drawBoard()` to print out the board to the player. Why do we do this now? Because either the computer was selected to make the first move of the game, in which case we should display the original starting

move, we check if there exist any possible moves the human player can make. If `getValidMoves()` returns an empty list, then there are no possible moves. That means the game is over, and we should break out of the `while` loop that we are in.

Otherwise, there is at least one possible move the player should make, so we should set `turn` to `'player'`

The game is now over and the winner has been declared. We should call our `playAgain()` function, which returns `True` if the player typed in that they want to play another game. If `playAgain()` returns `False` (which makes the `if` statement's condition `True`), we break out of the `while` loop (the one that started on line 248), and since there are no more lines of code after this `while`-block, the program terminates.

Otherwise, `playAgain()` has returned `True` (which makes 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.

Summary: Reviewing the Reversi Game

The AI may seem almost unbeatable, but this isn't because the computer is very smart. The strategy it follows is very 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 take us a long time to figure out how many tiles would be flipped for every possible valid move we could make. But calculating this for the computer is very simple. The computer isn't smarter than us, it's just much faster!

This game is very similar to Sonar because it makes use of a grid for a board. It is also like the Tic Tac Toe game because there is an AI that plans out the best move for it to take. This chapter only introduced one new concept: using the `bool()` function and the fact 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 that you already knew! You don't have to know very much about programming in order 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 just text and keyboard input.

Topics Covered In This Chapter:

- Simulations
- Percentages
- Pie Charts
- Integer Division
- The `round ()` Function

"Computer vs. Computer" Games

The Reversi AI algorithm was very simple, but it beats me almost every time I play it. This is because the computer can process instructions very fast, so checking each possible position on the board and selecting the highest scoring move is easy for the computer. If I took the time to look at every space on the board and write down the score of each possible move, it would take a long time for me to find the best move.

Did you notice that our Reversi program in Chapter 14 had two functions,

We are going to make three new programs, each based on the Reversi program in the last chapter. We will make changes to `reversi.py` to create *AI`Sim1.py`*. Next we will make changes to *AI`Sim1.py`* to create *AI`Sim2.py`*. And finally, we will make changes to *AI`Sim2.py`* to for *AI`Sim3.py`*

```

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

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

g code. The additions are in bold, and some lines have been removed.

changing the file, save it as *AI_Sim2.py*.

g, you can always download the *AI_Sim2.py* source code from the book's
entwithpython.com/chapter16.

```
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.' %
```

```
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 algorithm does have a random part, your run might not have the exact same numbers as above.

Printing things out to the screen slows the computer down, but now that we have removed that code, the computer can run an entire game of Reversi in about a second or two. Think about it. Each time our program printed out one of those lines, it ran through an entire game (which is about fifty or sixty moves, each move carefully checked to be the one

Try entering the following code into the interactive shell:

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 our 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. 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, we divide the number of wins or ties by the total number of games. We convert `numGames` to a float to ensure we do not use integer division in our calculation. Then we multiply the result by 100. However, we may end up with a number like `66.66666666666667`. So we 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,

```

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
291.     tiles.
292.     possibleMoves = getValidMoves(board, tile)
293.
294.     # randomize the order of the possible moves
295.     random.shuffle(possibleMoves)
296.
297.     # Go through all the possible moves and remember the
298.     best scoring move
299.     worstScore = 64
300.     for x, y in possibleMoves:
301.         dupeBoard = getBoardCopy(board)
302.         makeMove(dupeBoard, tile, x, y)
303.         score = getScoreOfBoard(dupeBoard)[tile]
304.         if score < worstScore:
305.             worstMove = [x, y]
306.             worstScore = score
307.
308.     return worstMove
309.
310. def getCornerWorstMove(board, tile):
311.     # Return a corner, a space, or the move that flips the
312.     least number of tiles.
313.     possibleMoves = getValidMoves(board, tile)
314.
315.     # randomize the order of the possible moves
316.     random.shuffle(possibleMoves)
317.
318.     # always go for a corner if available.
319.     for x, y in possibleMoves:
320.         if isOnCorner(x, y):
321.             return [x, y]
322.
323.     return getWorstMove(board, tile)
324.
325. print('Welcome to Reversi!')

```

How the AISim3.py Code Works

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 50 points. O scored 14 points.

Game #1: X scored 38 points. O scored 8 points.



Topics Covered In This Chapter:

- Software Libraries
- Installing Pygame
- Graphical user interfaces (GUI)
- Drawing primitives
- Creating a GUI window with Pygame
- Color in Pygame
- Fonts in Pygame
- Aliased and Anti-Aliased Graphics
- Attributes
- The `pygame.font.Font` Data Type
- The `pygame.Surface` Data Type
- The `pygame.Rect` Data Type
- The `pygame.PixelArray` Data Type
- Constructor Functions
- The `type()` Function
- Pygame's Drawing Functions
- The `blit()` Method for Surface Objects
- Events
- The Game Loop
- 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. This is simple, and an easy way to learn programming. But in this chapter, we will make some more exciting games with advanced graphics and sound using the Pygame library. Chapters 17, 18, and 19 will teach

you how to use the Pygame library to make games with graphics, animation, and sound. In these chapters we'll find source code for simple programs that are not games but demonstrate the Pygame concepts we've learned. Chapter 20 will present the source code for a complete Pygame game using all the concepts you've learned.

A **software library** is code that is not meant to be run by itself, but included in other programs to add new features. By using a library a programmer doesn't have to write the entire program, but can make use of the work that another programmer has done before them. Pygame is a software library that has modules for graphics, sound, and other features that games commonly use.

Installing Pygame

Pygame does not come with Python. Like Python, Pygame is available for free. You will have to download and install Pygame, which is as easy as downloading and installing the Python interpreter. In a web browser, go to the URL <http://pygame.org> and click on the "Downloads" link on the left side of the web site. This book assumes you have the Windows operating system, but Pygame works the same for every operating system. You need to download the Pygame installer for your operating system and the version of Python you have installed (3.1).

You do not want to download the "source" for Pygame, but rather the Pygame for your operating system. For Windows, download the *pygame-1.9.1.win32-py3.1.msi* file. (This is Pygame for Python 3.1 on Windows. If you installed a different version of Python (such as 2.5 or 2.4) download the .msi file for your version of Python.) The current version of Pygame at the time this book was written is 1.9.1. If you see a newer version on the website, download and install the newer Pygame. For Mac OS X and Linux, follow the directions on the download page for installation instructions.

pygameHelloWorld.py

This code can be downloaded from <http://inventwithpython.com/pygameHelloWorld.py>
If you get errors after typing this code in, compare it to the book's code with the online diff tool at <http://inventwithpython.com/diff> or email the author at al@inventwithpython.com

```
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)
```

```

    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 GUI window appear which looks like Figure 17-2.

What is nice about using a GUI instead of a console is that the text can appear anywhere in the window, not just after the previous text we have printed. The text can be any color or size.

One thing you may notice is that Pygame uses a lot of tuples instead of lists. Remember that tuples are almost the same as lists (they can contain other values) except they are typed with parentheses (and), instead of square brackets [and]. The main difference is that once you create a tuple, you cannot change, add, or remove any values in the tuple. For technical reasons, knowing that the contents of the tuple never change allows Python to handle this data more efficiently, which is why Pygame uses tuples instead of lists.

Figure

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 that you know which module the `exit()` is in.)

The `pygame.init()` Function

```
4. # set up pygame
5. pygame.init()
```

The Pygame software library has some initial code that needs to be run before we can use it. All Pygame programs must run this code by calling the `pygame.init()` after

Colors in Pygame

```
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)
```

There are three primary colors of light: red, green and blue. By combining different amounts of these three colors you can form any other color. In Python, we represent colors with tuples of three integers. The first value in the tuple is how much red is in the color. A value of 0 means there is no red in this color, and a value of 255 means there is a maximum

Attributes

```
24. textRect.centerx = windowSurface.get_rect().centerx
25. textRect.centery = windowSurface.get_rect().centery
```

The `pygame.Rect` data type (which we will just call `Rect` for short) makes working with rectangle-shaped things easy. To create a new

The great thing about `Rect` objects is that if you modify any of these variables, all the

Functions that have the same name as their data type and create objects or values of this data type are called

stored in `windowSurface` with the color white. The `fill()` function will completely cover the entire surface with the color we pass as the parameter. (In this case, we pass `BLACK` to make the background black.)

An important thing to know about Pygame is that the window on the screen will not change when we call the `fill()` method or any of the other drawing functions. These will draw on the `Surface` object, but the `Surface` object will not be drawn on the user's screen until the `pygame.display.update()` function is called. This is because drawing on the `Surface` object (which is stored in the computer's memory) is much faster than drawing to the computer screen. It is much more efficient to draw onto the screen once and only a2(o t)8(n. I)8 and onlyv(ny 8 ao)9(ur) [(dr)3(a)4(w)2(i)-2()4(2)3(unc)4(t)-2(i)-2-2(s)-1(8)-2(o d

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 `pygame.draw.line()` function will draw a line on the `Surface` object that you provide. Notice that the last parameter (the width of the line) is optional. 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 `pygame.draw.circle()` Function

```
38. # draw a blue circle onto the surface
39. pygame.draw.circle(windowSurface, BLUE, (300, 50), 20, 0)
```

The `pygame.draw.circle()` function will draw a circle on the `Surface` object you provide. The third parameter is for the X and Y coordinates of the center of the circle as a tuple of two ints. The fourth parameter is an `int` for the radius (that is, size) of the circle in pixels. A width of 0 means that the circle will be filled in.

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 will draw an ellipse. It is similar to the `pygame.draw.circle()` function, except that instead of specifying the center of the circle, a tuple of four ints is passed for the left, top, width, and height of the ellipse.

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 ints for the left, top, width, and height of the rectangle. Instead of a tuple of four ints for the third parameter, you can also pass a `Rect` object. In line 45, we want the rectangle we draw to be 20 pixels around all the sides of the text. This is why we want the drawn rectangle's left and top to be the left and top of `textRect` minus 20. (Remember, we subtract because coordinates decrease as you go left and up.) And the width and height will be equal to the width and height of the `textRect` plus 40 (because the left and top were moved back 20 pixels, so we 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
```

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!" text (which was drawn on the `Surface` object stored in the `text` variable) and draws it to the `Surface` object stored in the `windowSurface` variable.

Remember that the `text` object had the "Hello world!" text drawn on it on line 22 by the `render()` method. `Surface` objects are just stored in the computer's memory (like any other variable) and not drawn on the screen. The `Surface` object in `windowSurface` is drawn on the screen (when we call the `pygame.display.update()` function on line 56 below) because this was the `Surface` object created by the `pygame.display.set_mode()` function.

The second parameter to `blit()` specifies where on the `windowSurface` surface the `text` surface should be drawn. We will just pass the `Rect` object we got from calling `text.get_rect()` (which was stored in `textRect` on line 23).

The `pygame.display.update()` Function

```
55. # draw the window onto the screen
56. pygame.display.update()
```

In Pygame, nothing is drawn to the screen until the `pygame.display.update()` function is called. This is done because drawing to the screen is a slow operation for the computer compared to drawing on the `Surface` objects while they are in memory. You do not want to draw to the screen after each drawing function is called, but only draw the screen once after all the drawing functions have been called.

You will need to call `pygame.display.update()` each time you want to update the screen to display the contents of the `Surface` object returned by `pygame.display.set_mode()`. (In this program, that object is the one stored in `windowSurface`.) This will become more important in our next program which covers animation.

Events and the Game Loop

In our previous games, all of the programs print out 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. Pygame programs do not work this way. Instead, Pygame programs are constantly running through a loop called the game loop. (In this program, we execute all the lines of code in the game loop about one 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 values of the `pygame.event.Event` data type that are generated by Pygame whenever the user

know that we should run any code that we want to happen to stop the program. You could choose to ignore the `QUIT` event entirely, but that may cause the program to be confusing to the user.

The `pygame.quit()` Function

```
62.         pygame.quit()
63.         sys.exit()
```

If the `QUIT` event has been generated, then we can know that the user has tried to close the window. In that case, we should call the exit functions for both Pygame (`pygame.quit()`) and Python (`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 they are more complicated, the Pygame programs can also be much more fun and engaging than our previous 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. In order 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. By drawing new blocks that are located a little bit differently than the blocks before, we can make it look like the blocks are moving around the screen.

The Animation Program's Source Code

Type the following program into the file editor and save it as *animation.py*. You can also download this source code from <http://inventwithpython.com/chapter17>.

animation.py

This code can be downloaded from <http://inventwithpython.com/animation.py>
If you get errors after typing this code in, compare it to the book's code with the online diff tool at <http://inventwithpython.com/diff> or email the author at al@inventwithpython.com

```
1. import pygame, sys, time
2. from pygame.locals import *
TJ /T1_1 12 Tf 109.(2)-2(.1.1.1.)3(w)-3(i3)-2(
```

```

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: The Animation program.

How the Animation Program Works

In this program the size of the window's width and height is used for more than just the call to `set_mode()`. We will use a constant variables to make the program more readable. Remember, readability is for the benefit of the programmer, not the computer. If we ever want to change the size of the window, we only have to change lines 8 and 9.

If we did not use the constant variable, we would have to change ever occurrence of the int value 400. If any unrelated values in the program were also 400, we might think it was for the width or height and also accidentally change it too. This would put a bug in our program. 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')
```

For this program, we will set the caption at the top of the window to 'Animation' with a call to `pygame.display.set_caption()`.

```
13. Setting Up Constant Variables for Direction
14. # set up direction variables
15. DOWNLEFT = 1
16. DOWNRIGHT = 3
17. UPLEFT = 7
h-8(8o)
```

We will 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)
```


positions, colors, and directions.

```
31. blocks = [b1, b2, b3]
```

On line 31 we put all of these data structures in a list, and store the list in a variable named `rectangles`.

`rectangles` is a list. `rectangles[0]` would be the dictionary data structure in `r1`. `rectangles[0]['color']` would be the 'color' key in `r1` (which we stored the value in RED in), so the expression `rectangles[0]['color']` would evaluate to `(255, 0, 0)`. In this way we can refer to any of the values in any of the block data structures by starting with `rectangles`.

Running the Game Loop

```
33. # run the game loop
34. while True:
```

Inside the game loop, we want to move all of the blocks around the screen in the direction that they are going, then bounce the block if they have hit a wall, then draw all of the blocks to the `windowSurface` surface, and finally call `pygame.display.update()` to draw the surface to the screen. Also, we will call `pygame.event.get()` to check if the QUIT event has been generated by the user closing the window.

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, so we will skip its explanation and go on to line 44.

```
41.     # draw the black background onto the surface
42.     windowSurface.fill(BLACK)
```

Before we draw any of the blocks on the `windowSurface` surface, we want to fill the entire surface with black so that anything we previously drew on the surface is covered. Once we have blacked out the entire surface, we can redraw the blocks with the code below.

Moving Each Block

```
44.     for b in blocks:
```

We want to update the position of each block, so we must loop through the `rectangles` list and perform the same code on each block's data structure. Inside the loop, we will refer to the current block as simply `r` 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 that we want to set the `left` and `top` attributes to depends on the direction the block is moving. Remember that the X-coordinates start at 0 on the very left edge of the window, and increase as you go right. The Y-coordinates start at 0 on the very top of the window, and increase as you go down. So if the direction of the block (which, remember, is stored in the `'dir'` key) is either `DOWNLEFT` or `DOWNRIGHT`, we want to increase the `top` attribute. If the direction is `UPLEFT` or `UPRIGHT`, we want to decrease the `top` attribute.

If the direction of the block is `DOWNRIGHT` or `UPRIGHT`, we want to *increase* the `left` attribute. If the direction is `DOWNLEFT` or `UPLEFT`, we want to *decrease* the `left` attribute.

We could have also modified `right` instead of the `left` attribute, or the `bottom` attribute instead of the `top` attribute, because Pygame will update the `Rect` object either way. Either way, we want to change the value of these attributes by the integer stored in `MOVESPEED`, which stores how many pixels over we will move the block.

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 we have moved the block, we want to check if the block has gone past the edge of

the window. If it has, we want to "bounce" the block, which in the code means set a new value for the block's `'dir'`

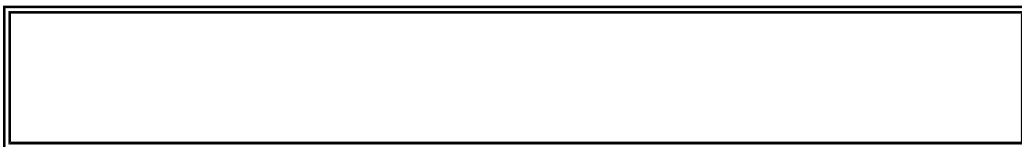
```
80.             if b['dir'] == DOWNRIGHT:
81.                 b['dir'] = DOWNLEFT
82.             if b['dir'] == UPRIGHT:
83.                 b['dir'] = UPLEFT
```

This code is similar to the previous pieces of code, but it checks if the block has moved past the rightmost 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 we have moved the block (and set a new direction if the block has bounced off the window's edges), we want to draw it on the `windowSurface` surface. We can draw this using the `pygame.draw.rect()` function. We pass `windowSurface`, because that is the `Surface` object we want to draw on. We pass the `b['color']` value, because this is the color we want to use. Then we pass `b['rect']`, because that `Rect` object has the information about the position and size of the rectangle we want to draw.



is very similar. The `blocks` variable held a list of data structures representing things 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 our program can know when the player presses any key on the keyboard. We will also learn of a concept called collision detection, which is used in many graphical computer games.

Topics Covered In This Chapter:

Collision Detection

the previous chapter for an explanation of that code.) We will use a list of `pygame.Rect` objects to represent the food squares. Each `pygame.Rect` object in the list represents a single food square. On each iteration through the game loop, our program will read each `pygame.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 `pygame.Rect` 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 just 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.

Type the following into a new file and save it as `collisionDetection.py`. If you don't want to type all of this code, you can download the source from the book's website at <http://inventwithpython.com/chapter18>.

collisionDetection.py

This code can be downloaded from <http://inventwithpython.com/collisionDetection.py>. If you get errors after typing this code in, compare it to the book's code with the online diff tool at <http://inventwithpython.com/diff> or email the author at al@inventwithpython.com

```
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 >
17.         rect.top) and (y < rect.bottom):
18.         return True
19.     else:
20.         return False
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 ( )-2( )]TJ T* [(5)-2(8)-2(.)-6.py QQQQUIT

```


When you run this code, this is what the program looks like. The white square (the bouncer) will bounce around the window, and when it collides with the green squares (the food) will disappear from the screen.

Figure 18-1: 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 Animation9-358 l 31o3ramy-13.32 Td [

Figure 18-2: Examples of intersecting rectangles (on the left) and rectangles that do not intersect (on the right).

We will make a single function that is passed two `pygame.Rect` objects. The function, `doRectsOverlap()`, will return `True` if they do and `False` if they don't.

There is a very simple rule we can follow to determine if rectangles intersect (that is, 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 we know that the two rectangles have collided.

We don't want to repeat the code that checks all four corners for both `rect1` and `rect2`, so instead we use `a` and `b` on lines 7 to 10. The `for` loop on line 5 uses the multiple assignment trick so that 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`.

We do this because then we only have to type the code for the `if` statement on line 7 once. This is good, because this is a very long `if` statement. The less code we have to type for our program, the better.

```
13.     return False
```

If we never return `True` from the previous `if` statements, then none of the eight corners we checked are in the other rectangle. In that case, the rectangles did not collide and we return `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 used by the `doRectsOverlap()` function. `isPointInsideRect()` will return

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.



first two parameters for `pygame.Rect()` are the XY coordinates of the top left corner. We want the random coordinate to be between 0 and the size of the window minus the size of the food square. If we had the random coordinate between 0 and the size of the window, then the food square might be pushed outside of the window altogether. Look at the diagram in Figure 18-4.

The square on the left has an X-coordinate of its top left corner at 380. Because the food square is 20 pixels wide, the right edge of the food square is at 400. (This is because $380 + 20 = 400$.) The square on the right has an X-coordinate of its top left corner at 400. Because the food square is 20 pixels wide, the right edge of the food square is at 420, which puts the entire square outside of the window (and not viewable to the user).

The third parameter for `pygame.Rect()` is a tuple that contains the width and height of the food square. Both the width and height will be equal to 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 very similar to lines 44 to 83 of our animation program in the last chapter, so we will not go over them again here.

```
111.     # draw the bouncer onto the surface
112.     pygame.draw.rect(windowSurface, WHITE, bouncer
                        ['rect'])
```

After moving the bouncer, we now want to draw it on the window in its new position. We call the `pygame.draw.rect()` function to draw a rectangle. The `windowSurface` passed for the first parameter tells the computer which `pygame.Surface` object to draw the rectangle on. The `WHITE` variable, which has `(255, 255, 255)` stored in it, will tell the computer to draw a white rectangle. The `pygame.Rect` object stored in the `bouncer` dictionary at the `'rect'` key tells the position and size of the rectangle to draw. This is all the information needed to draw a white rectangle on `windowSurface`.

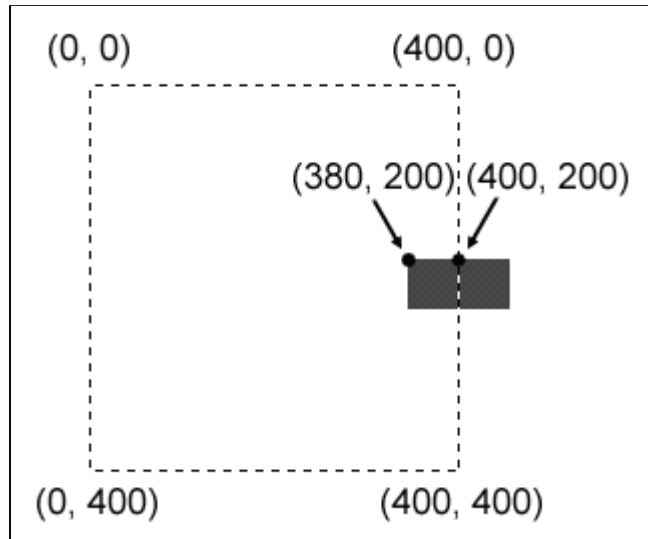


Figure 18-4: For a 20 by 20 rectangle, having the top left corner at (400, 200) in a 400 by 400 window would place the rectangle outside of the window. To be inside, the top left corner should be at (380, 200) instead.

Remember, we are not done drawing things on the `windowSurface` object yet. We still need to draw a green square for each food square in the `foods` list. And we are just "drawing" rectangles on the `windowSurface` object. This `pygame.Surface` object is only inside the computer's memory, which is much faster to modify than the pixels on the screen. The window on the screen will not be updated until we call the `pygame.display.update()` function.

Colliding with the Food Squares

```
114.     # check if the bouncer has intersected with any food
        squares.
115.     for food in foods[:]:
```

Before we draw the food squares, we want to see if the bouncer has overlapped any of the food squares. If it has, we will remove that food square from the `foods` list. This way, the computer 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 will be stored inside a variable called `food` (singular).

Don't Add to or Delete from a List while Iterating Over It

Notice that there is something slightly different with this `for` loop. If you look carefully at line 116, we are not iterating over `foods` but actually over `foods[:]`. Just as `foods[:2]` would return a copy of the list with the items from the start and up to (but not including) the item at index 2, and just as `foods[3:]` would return 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 our `getBoardCopy()` function in the Tic Tac Toe game.)

Why would we want to iterate over a copy of the list instead of the list itself? It is because we cannot add or remove items from a list while we 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 for you if you tried to count the number of jelly beans in a jar while someone was adding or removing jelly beans. But if we iterate over a copy of the list (and the copy never changes), then 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 our `doRectsOverlap()` function that we defined earlier comes in handy. We pass two `pygame.Rect` objects to `doRectsOverlap()`: the bouncer and the current food square. If these two rectangles overlap, then `doRectsOverlap()` will return `True` and we will remove the overlapping food squares from `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 very similar to how we drew the white square for the player. We will loop through each food square in the `foods` list, and then draw the rectangle onto the `windowSurface` surface. This demonstration of collision detection is fairly easy. This program was very similar to our bouncing program in the previous chapter, except now the bouncing square will "eat" the other squares as it passes over them.

These past few programs are interesting to watch, but the user does not get to actually control anything. In this next program, we will learn how to get input from the keyboard. Keyboard input is handled in Pygame by using events.

The Keyboard Input Program's Source Code

Start a new file and type in the following code, then save it as *pygameInput.py*.

```
pygameInput.py
Thi
```

```

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 looks identical to the collision detection program earlier in this chapter. But in this program, the bouncer only moves around when we hold down keys on the keyboard. Holding down the "W" key moves the bouncer up. The "A" key moves the bouncer to the left and the "D" key moves the bouncer to the right. The "S" key moves the bouncer down. You can also move the bouncer by holding down the arrow keys on the keyboard. The user can also use the keyboard's arrow keys.



Events and Handling the KEYDOWN Event

Table 18-1: Events, and what causes them to be generated.

Event	Description												
QUIT	Generated when the user closes with window.												
KEYDOWN	Generated when the user pressed down a key. Has a <code>key</code> attribute that tells which key was pressed. Also has a <code>mod</code> 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 <code>key</code> and <code>mod</code> attribute that are similar to those for KEYDOWN.												
MOUSEMOTION	Generated whenever the mouse moves over the window. Has a <code>pos</code> attribute that returns tuple (x, y) for the coordinates of where the mouse is in the window. The <code>rel</code> 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 <code>rel</code> will be (-4, 0). The <code>buttons</code> 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 is 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 <code>pos</code> attribute which is an (x, y) tuple for the coordinates of where the mouse was when the button was pressed. There is also a <code>button</code> attribute which is an integer from 1 to 5 that tells which mouse button was pressed: <table border="1" data-bbox="608 1255 1208 1526" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Value of <code>button</code></th> <th>Mouse Button</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Left button</td> </tr> <tr> <td>2</td> <td>Middle button</td> </tr> <tr> <td>3</td> <td>Right button</td> </tr> <tr> <td>4</td> <td>Scroll wheel moved up</td> </tr> <tr> <td>5</td> <td>Scroll wheel moved down</td> </tr> </tbody> </table>	Value of <code>button</code>	Mouse Button	1	Left button	2	Middle button	3	Right button	4	Scroll wheel moved up	5	Scroll wheel moved down
Value of <code>button</code>	Mouse Button												
1	Left button												
2	Middle button												
3	Right button												
4	Scroll wheel moved up												
5	Scroll wheel moved down												
MOUSEBUTTONUP	Generated when the mouse button is released. This has the same attributes as MOUSEBUTTONDOWN												

The code to handle the key press and key release events is below. But at the start of the program, we will set all of these variables to `False`.

```
44.         if event.type == KEYDOWN:
```

Pygame has another event type called `KEYDOWN`. On line 41, we check if the `event.type` attribute is equal to the `QUIT` value to check if we should exit the program. But there are other events that Pygame can generate. A brief list of the events that could be returned by

(' a '), we make the left arrow key and the A key do the same thing. You may notice that the W, A, S, and D keys are all used as alternates for changing the movement variables. This is because some people may want to use their left hand to press the WASD keys instead of their right hand to press the arrow keys. Our program offers them both!

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 we want to terminate the program. Remember, in Pygame you must call the `pygame.quit()` function before calling the `sys.exit()` function. We want to do this when the user releases the Esc key, not when


```
72.             player.left = random.randint(0,
        WINDOWWIDTH - player.width)
```

We will also add teleportation to our game. If the user presses the "X" key, then we 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 (though 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 clicks a mouse button somewhere in our window, and releases the mouse button. The `pos` attribute in the `Event` object is set to a tuple of two integers for the XY coordinates. On line 75, the X-coordinate is stored in `event.pos[0]` and the Y-coordinate is stored in `event.pos[1]`. We will create 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, a new food square will be displayed on the screen.

Moving the Bouncer 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
```

We have set the movement variables (`moveDown`, `moveUp`, `moveLeft`, and `moveRight`) to

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 our previous Collision Detection program, we had our own function to check if one rectangle had collided with another. That function was included in this book so that you could understand how the code behind collision detection works. In this program, we 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. This is the exact same behavior as the `doRectsOverlap()` function in our previous Collision Detection program.

```
110.     mainClock.tick(40)
```

The rest of the code is similar to the code in the Input program is similar to the earlier Collision Detection program: draw the food squares and the player squares to the `windowSurface` surface, occasionally add a new food square at a random location to the `foods` list, check if the player square has collided with any of the food squares, and call `mainClock.tick(40)` to make the program run at an appropriate speed.

Summary: Collision Detection and Pygame Input

This chapter introduced the concept of collision detection, which is used in most graphical games. Detecting collisions between two rectangles is easy: we just 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

use pictures and images instead of simple drawing primitives. The next chapter will tell you how to load images and draw them on the screen. We will also learn how to play sounds and music for the player to hear.



Topics Covered In This Chapter:

- Image and Sound 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 shapes in different colors on the screen. 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. Here are some example sprites:

Figure 19-1: Some examples of sprites.

This is an example of sprites 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 the background. Notice that we can flip the sprite

large sprite.

```
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')
```

```
52.         if event.key == K_LEFT or event.key == ord
           ('
```



```
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)
```

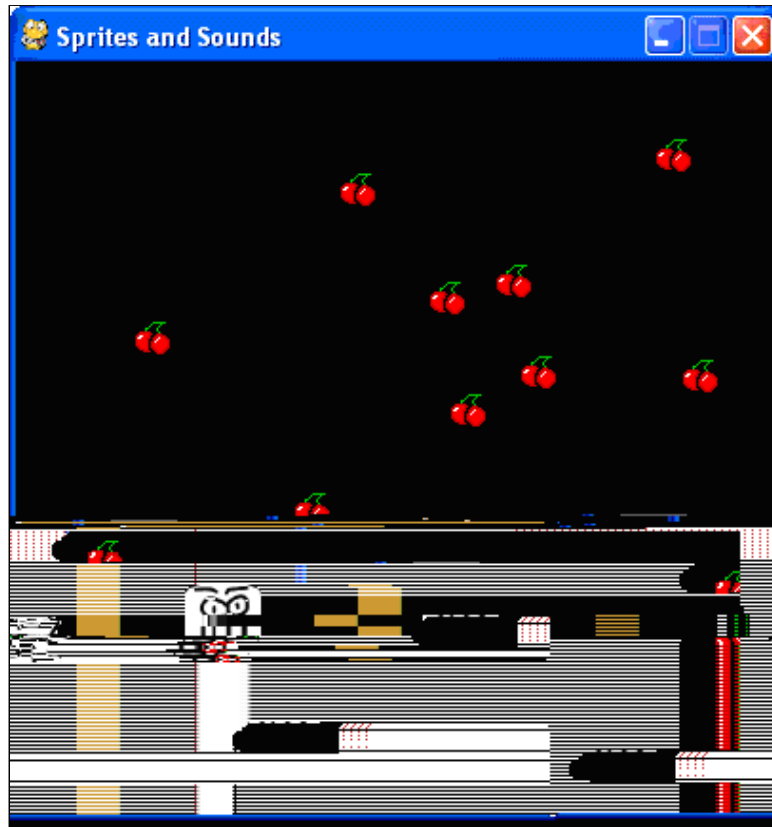


Figure 19-3: The Sprites and Sounds game.

Setting Up the Window and the Data Structure

Most of the code in this program was explained in the previous chapter, so we will only focus 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. 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 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, just the player's size and location. At the beginning of the program, the top left corner of the player will be located at (300, 100) and the player will have a height and width of 40 pixels to start.

The second variable that represents the player will be `playerImage`. The `pygame.image.load()` function is passed a string of the filename of the image to load. The return value of `pygame.image.load()` is a `Surface` object that has the image in the image file drawn on its surface. We store this `Surface` object inside of `playerImage`.

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 that you have the *player.png* and *cherry.png* file in the same directory as the *spritesAndSounds.py* file, otherwise Pygame will not be able to find them and will give an error.

The `Surface` objects that are stored in `playerImage` and `foodImage` are the same as the `Surface`

Toggling the value in `musicPlaying` will ensure that the next time the user presses the M key, it will do the opposite of what it did before.

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. "Blitting" is the process of drawing the contents of one `Surface` object to another `Surface` object. In this case, we want to draw the sprite of the player onto the window's `Surface` object (which is stored in `windowSurface`). (Also remember that the surface used to display on the screen is the `Surface` object that is returned by `pygame.display.set_caption()`.)

The second parameter to the `blit()` method is a `Rect` object that specifies where 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,
117.                                   player.width + 2, player.height + 2)
117.             playerStretchedImage = pygame.transform.scale
118.                 (playerImage, (player.width, player.height))
118.             if musicPlaying:
119.                 pickUpSound.play()
```

This code is similar to the code in the previous programs. But here we are adding a couple of new lines. We want to call the `play()` method on the `Sound` object stored in the `pickUpSound` variable. But we only want to do this if `musicPlaying` is set to `True` (which tells us that the sound turned on).

When the player eats one of the cherries, we are going to enlarge the size of the player by two pixels in height and width. On line 116, we create a new `Rect` object to store in the `player` variable which will have the samey(s)-1(i)-2(z)4(e)-6(s)-1anes thel `Rect`

`playerStretchedImage`. Stretching an image often distorts it a little. If we keep restretching a stretched image over and over, the distortions add up quickly. But by stretching the original image to the new size, we only distort the image once. This is why we pass

Topics Covered In This Chapter:

- The `pygame.FULLSCREEN` flag
- Pygame Constant Variables for Keyboard Keys
- The `move_ip()` Method for `Rect` objects
- The `pygame.mouse.set_pos()` Function

Review of the Basic Pygame Data Types

Let's review some of the basic data types used in the Pygame library:

`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 `collidect()` method to check if they are intersecting 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 will be displayed on the user's screen.

Remember that `Surface` have things drawn on them, but we cannot see this because it only exists in the computer's memory. We can only see a `Surface` object when it is "blitted" (that is, drawn) on the screen. This is just the same as it is with any other piece of data. If you think about it, you cannot see the string that is stored in a variable until the variable is printed to the screen.

`pygame.event.Event` - The `Event` data type in 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 `Event` objects. You can check what type of event the `Event` object is by checking its type attribute. `QUIT`, `KEYDOWN`, and `MOUSEBUTTONDOWN` are examples of some event types.

`pygame.font.Font` - The `pygame.font` module has the `Font` data type which represent the typeface used for text in Pygame. You can create a `Font` object by calling the `pygame.font.SysFont()` constructor function. The arguments to pass 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. For example, the common function call to create a `Font` object is `pygame.font.SysFont(None, 48)`.

`pygame.time.Clock` - The `Clock` object in the `pygame.time` module are very helpful for keeping our games from running as fast as possible. (This is often too fast for the player to keep up with the computer, and makes the games not fun.) 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. Normally we use 40 fps. Notice that the `pygame.time` module is a different module than the `time` module which contains the `sleep()` function.

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://inventwithpython.com/resources>.

Dodger's Source Code

You can download this code from the URL <http://inventwithpython.com/chapter20>.

dodger.py

This code can be downloaded from <http://inventwithpython.com/dodger.py>
If you get errors after typing this code in, compare it to the book's code with the online diff tool at <http://inventwithpython.com/diff> or email the author at al@inventwithpython.com

```
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()
```

```

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 -
74. 50)
75.     moveLeft = moveRight = moveUp = moveDown = False
76.     reverseCheat = slowCheat = False
77.     baddieAddCounter = 0
78.     pygame.mixer.music.play(-1, 0.0)
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(BACKGROUND_COLOR)
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.
```

Figur

is a color for the background. However, the line `windowSurface.fill`
(`BACKGROUND_COLOR`) is not as clear what the argument being passed means.

We can also easily change some simple aspects about our game without having the

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
```

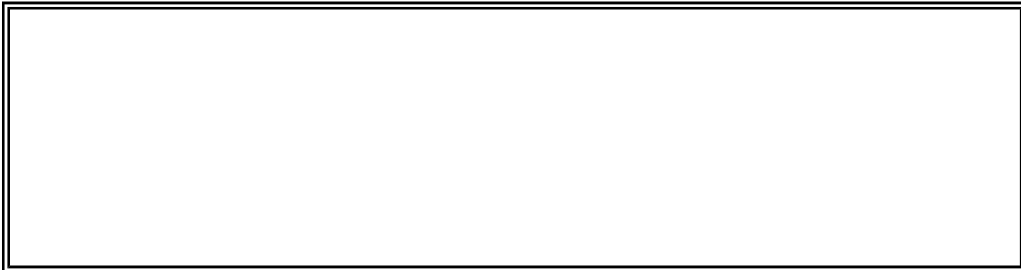
The PLAYERMOVERATE will store the number of pixels the player's character moves in the window on each iteration through the game loop (if the character is moving). By increasing this number, you can increase the speed the character moves. If you set PLAYERMOVERATE to 0, then the player's character won't be able to move at all (the player would move 0 pixels per iteration). This wouldn't be a very fun game.

Defining Functions

We will create several functions for our game. By putting code into functions, we can avoid having to type the same code several times. We can


```
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 t3280r wiils mfa4(ny dir)-2(f32(f)7(ea)4(r)33280rn(t)-2((s)1(t)83280rp(s)1(. F)62(i)-2



arguments for `pygame.display.set_mode()` are not two integers but a tuple of two integers.

On line 46, the caption of the window is set to the string `'Dodger'`. This caption will appear in the title bar at the top of the window.

In our game, we do not want the mouse cursor (the mouse cursor is the arrow that moves around the screen when we move the mouse) to be visible. This is because we want the mouse to be able to move the player's character around the screen, and the arrow cursor would get in the way of the character's image on the screen. We pass `False` to tell Pygame to make the cursor invisible. If we wanted to make the cursor visible again at some point in the program, we could call `pygame.mouse.set_visible(True)`.

Fullscreen Mode

The `pygame.display.set_mode()` function has a second, optional parameter that you can pass to it. The value you can pass for this parameter is `pygame.FULLSCREEN`.

```
53. gameOverSound = pygame.mixer.Sound('gameover.wav')
54. pygame.mixer.music.load('background.mid')
```

Next we want to create the `Sound` objects and also set up the background music. The background music will constantly be playing during the game, but `Sound` objects will only be played when we specifically want them to. In this case, the `Sound` object will be played when the player loses the game.

You can use any `.wav` or `.mid` file for this game. You can download these sound files from this book's website at the URL <http://inventwithpython.com/resources>. Or you can use your own sound files for this game, as long as they have the filenames of `gameover.wav` and `background.mid`. (Or 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



```
(WINDOWWIDTH / 3) - 30, (WINDOWHEIGHT / 3) + 50)
64. pygame.display.update()
65. waitForPlayerToPressKey()
```

On lines 62 and 63, we call our `drawText()` function and pass it five arguments: 1) the string of the text we want to appear, 2) the font that we want the string to appear in, 3) the `Surface` object onto which to render the text, and 4) and 5) the X and 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 we call it. This shortens our program and makes it easier to find bugs since there is less code to check.

The `waitForPlayerToPressKey()` function will pause the game by entering into a loop that checks for any `KEYDOWN` events. Once a `KEYDOWN` event is generated, the execution breaks out of the loop and the program continues to run.



dba6m BT 0 02m 813 5

game world on

```
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`, then we know that the player has pressed down a key. The `Event` object for keyboard events will also have a `key` attribute that is set to the numeric ASCII value of the key pressed. The `ord()` function will return the ASCII value of the letter passed to it.

For example, on line 87, we can check if the event describes the "z" key being pressed down by checking if `event.key == ord('z')`. If this condition is `True`, then we want to set the `reverseCheat` variable to `True` to indicate that the reverse cheat has been activated. We will also check if the "x" key has been pressed to activate the slow cheat in a similar way.

Pygame's keyboard events always use the ASCII values of lowercase letters, not uppercase. What this means for your code is that you should always use `event.key ==`


```
from pygame.locals import * instead of import pygame.locals.
```

Noticed that pressing down on one of the arrow keys not only sets one of the movement variables 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.

Here is a list of commonly-used constant variables for the `key` attribute of keyboard-related `Event` objects:

```
104.         if event.type == KEYUP:
105.             if event.key == ord('z'):
106.                 reverseCheat = False
107.                 score = 0
```

```
110.                                     score = 0
```

The KEYUP event is created whenever the player stops pressing down on a keyboard key and it returns to its normal, up position. KEYUP objects with a type of KEYUP also have a key attribute just like KEYDOWN events.

On line 105, we check if the player has released the "z" key, which will deactivate the reverse cheat. In that case, we set reverseCheat to False and reset the score to 0. The




```
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.                     }
```

Line 138 will add the newly created baddie data structure to the list of baddie data structures. Our 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.

Line 151 moves the mouse cursor to the

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) will be ignored while the slow cheat is activated.

Removing the Baddies

```
162.         # Delete baddies that have fallen past the
           bottom.
163.         for b in baddies[:]:
```

After moving the baddies down the window, we want to remove any baddies that fell below the bottom edge of the window from the `baddies` list. Remember that while we are iterating through a list, we should not modify the contents of the list by adding or removing items. So instead of iterating through the `baddies` list with our `baddies` loop, we will iterate through a copy of the `baddies` list.

Remember that a list slice will evaluate a copy of a list's items. For example, `spam [2 : 4]` will return a new list with the items from index 2 up to (but not including) index 4. Leaving the first index blank will indicate that index 0 should be used. For example, `spam [: 4]` will return a list with items from the start of the list up to (but not including) the item at index 4. Leaving the second index blank will indicate that up to (and including) the last index should be used. For example, `spam [2 :]` will return a list with items from index 2 all the way to (and including) the last item in the list.

But leaving both indexes in the slice blank is a way to represent the entire list. The

Or you can find out more about Python by searching the World Wide Web. Go to the sea



Most of the programs in this book make use of the newer version 3 of Python. The Pygame games make use of Python 2 because the Pygame library is not yet compatible with Python 3. Python 3 corrects many of the faults with version 2 of the language, however, these changes can also make it impossible to run Python 3 programs with the Python 2 interpreter, and vice versa.

There are only a few changes between the two versions, and this appendix will go through the ones relevant to this book. Learning both Python 2 and 3 is fairly simple. All of the modules imported by the programs in this book (except for the pygame module) are part of the standard library and work with both Python 2 and Python 3.

In short, use Python 3 unless you need to use a library that is not yet compatible with version 3. Learning both Python 2 and 3 is easy because there are only a few changes between them.

The `print()` Function and the `print` statement

In Python 3, `print()` is a function just like `input()` or `len()`. The function version requires parentheses just like any other function call (although you can add parentheses to the `print` statement optionally).

The `print` statement in Python 2 will always print a newline character at the end of the string. To make it print a space instead, put a comma at the end of the `print` statement:

```
>>> # Python 2
```


Ack!

This appendix contains a list of all the statements, functions, and methods presented in

You may want to share the game programs you make with other people. Having other people play your games is a great way to show off your skills. However, they may not have Python installed on their computer. There is a way to run Python programs without installing the Python interpreter: You will have to compile your .py script into a .exe executable program.

Compiling source code means converting that source code into machine language, which is the programming language your computer understands. Programming in machine language is very long and tedious, and higher-level languages such as Python make programming easier.

This appendix will show you how to compile your .py Python files into .exe programs that can be run on Windows without having Python installed.

Step 1: Download and Install py2exe

much like how you installed Python. Just keep clicking the Next button until you reach the end.

After you have installed py2exe, you can make sure that the installation was successful by running the interactive shell and typing the following:

```
>>> import py2exe
>>>
```

If you do not see anything, then the installation was successful. If you see this error:

```
>>> import py2exe
>>> import py2exe
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named
```

script and setup.py. From that folder, type "c:\Python26\python.exe
setup.py py2exe". The first part (c:\Python26\python.exe

In the above case, the programmer used `=` (the assignment operator) instead of `==` (the equals comparator operator). Python never expects assignment statements where there should be a condition.

```
def foo(:
```

In the above case, the programmer forgot to match the ending `)` closing parenthesis.

```
def foo()
```

In the above case, the programmer forgot to put the colon at the end of the `def` statement. This can also happen with `for`, `while`, `if`, `elif`, and `else` statements.

ImportError: No module named raandom

This error shows up when you try to import a module that does not exist. Most likely, you have a typo in the module name. For example, you may have typed `raandom` instead of `random`.

Glossary

absolute value - The positive form of a negative number. For example, the absolute value of -2 is 2. The absolute value of a positive number is simply the positive number itself.

AI - see, artificial intelligence

algorithm - A series of instructions to compute something.

applications - A program that is run by an operating system. See also, program.

arguments - The values that are passed for parameters in a function call.

artificial intelligence - Code or a program that can intelligently make decisions (for example, decisions when playing a game) in response to user actions.

ASCII art

encrypting - To convert a message into a form that resembles garbage data, and cannot be understood except by someone who knows the cipher and key used to encrypt the message.

escape character - Escape characters allow the programmer to specify characters in Python that are difficult or impossible to type into the source code. All escape characters are preceded by a \ forward backslash character. For example, \n displays a newline character when it is printed.

evaluate - Reducing an expression down to a single value. The expression $2 + 3 + 1$ evaluates to the value 6.

execute - The Python interpreter executes lines of code, by evaluating any expressions or performing the task that the code does.

exit - When a program ends. "Terminate" means the same thing.

expression - Values and function calls connected by operators. Expressions can be evaluated down to a single value.

file editor - A program used to type in or change files, including files of Python source code. The IDLE program has a file editor that you use to type in your programs.

floating point numbers - Numbers with fractions or decimal points are not integers. The numbers 3.5 and 42.1 and 5.0 are floating point numbers.

flow chart - A chart that informally shows the flow of execution for a program, and the main events that occur in the program and in what order.

flow control statements - Statements that cause the flow of execution to change, often depending on conditions. For example, a function call sends the execution to the beginning of a function. Also, a loop causes the execution to iterate over a section of code several times.

flow of execution - The order that Python instructions are executed. Usually the Python interpreter will start at the top of a program and go down executing one line at a time. Flow control statements can move the flow of execution to different parts of code in the program.

function - A collection of instructions to be executed when the function is called. Functions also have a return value, which is the value that a function call evaluates to.

function call - A command to pass execution to the code contained inside a function, also passing arguments to the function. Function calls evaluate to the return value of the function.

global scope - The scope of variables outside of all functions. Python code in the global scope cannot see variables inside any function's local scope.

hard-coding - Using a value in a program, instead of using a variable. While a variable could allow the program to change, by hard-coding a value in a program, the value stays permanently fixed unless the source code is changed.

hardware - The parts of a computer that you can touch, such as the keyboard, monitor, case, or mouse. See also, software.

higher-level programming languages - Programming languages that humans can understand, such as Python. An interpreter can translate a higher-level language into machine code, which is the only language computers can understand.

IDLE - Interactive DeveLopment Environment. IDLE is a prrammiri -f 1

parameters named eggs and cheese.

pie chart - A circular chart that shows percentage portions as portions of the entire circle.

plaintext - The decrypted, human-readable form of a message.

player - A person who plays the computer game.

positive numbers - All numbers equal to or greater than 0.

pound sign

shell - see, interactive shell

simple substitution ciphers - A cipher where each letter is replaced by one and only one other letter.

slice - A subset of values in a list. These are accessed using the `:` colon character in between the square brackets. For example, if `spam` has the value `['a', 'b', 'c', 'd', 'e', 'f']`, then the slice `spam[2:4]` has the value `['c', 'd']`. Similar to a substring.

software - see, program

source code - The text that you type in to write a program.

statement - A command or line of Python code that does not evaluate to a value.

stepping - Executing one line of code at a time in a debugger, which can make it easier to find out when problems in the code occur.

string concatenation - Combining two strings together with the `+` operator to form a new string. For example, `'Hello ' + 'World!'` evaluates to the string `'Hello World!'`

string formatting - Another term for string interpolation.

string interpolation - Using conversion specifiers in a string as place holders for other values. Using string interpolation is a more convenient alternative to string concatenation. For example, `'Hello, %s. Are you going to %s on %s?'` `% (name, activity, day)` evaluates to the string `'Hello, Albert. Are you going to program on Thursday?'`, if the variables have those corresponding values.

string - A value made up of text. Strings are typed in with a single quote `'` or double `"` on either side. For example, `'Hello'`

substring - A subset of a string value. For example, if `spam` is the string `'Hello'`, then the substring `spam[1:4]` is `'ell'`. Similar to a list slice.

symbols - In cryptography, the individual characters that are encrypted.

syntax - The rules for how code is ordered in a programming language, much like grammar is made up of the rules for understandable English sentences.

syntax error - An error that occurs when the Python interpreter does not understand the code because the code is incomplete or in the wrong order. A program with a syntax error will not run.

terminate

About the Author



Albert Sweigart (but you can call him AI), is a software developer in San Francisco, California who enjoys bicycling, reading, volunteering, computer security, haunting coffee shops, and making useful software.

He is originally from Houston, Texas. He finally put his University of Texas at Austin computer science degree in a frame. He is, of course, an atheist. He is a friendly introvert, a cat person, and fears that he is losing brain cells over time. He laughs out loud when watching park squirrels, and people think he's a simpleton.

"Invent with Python" is his first book and will not be his last.

His web site and blog are located at <http://coffeeghost.net>

