Book Version 3.0.1, ISBN 978-1503212305

Attribution: Treasure chest icon by Victor Escorsin, Sonar icon by Pantelis Gkavos

If you’ve downloaded this book from a torrent, it’s probably out of date.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

For Caro, with more love

than I ever knew I had.

A Note to Parents and Fellow Programmers

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

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

The current crop of programming books fall into two categories. First, books that teach game creation software or a dumbed-down languages to make easy to the point that it is no longer programming. Or second, they taught programming like a mathematics textbook: all principles and concepts with little application given to the reader. This book takes a different approach: show the source code for games right up front and explain programming principles from the examples.

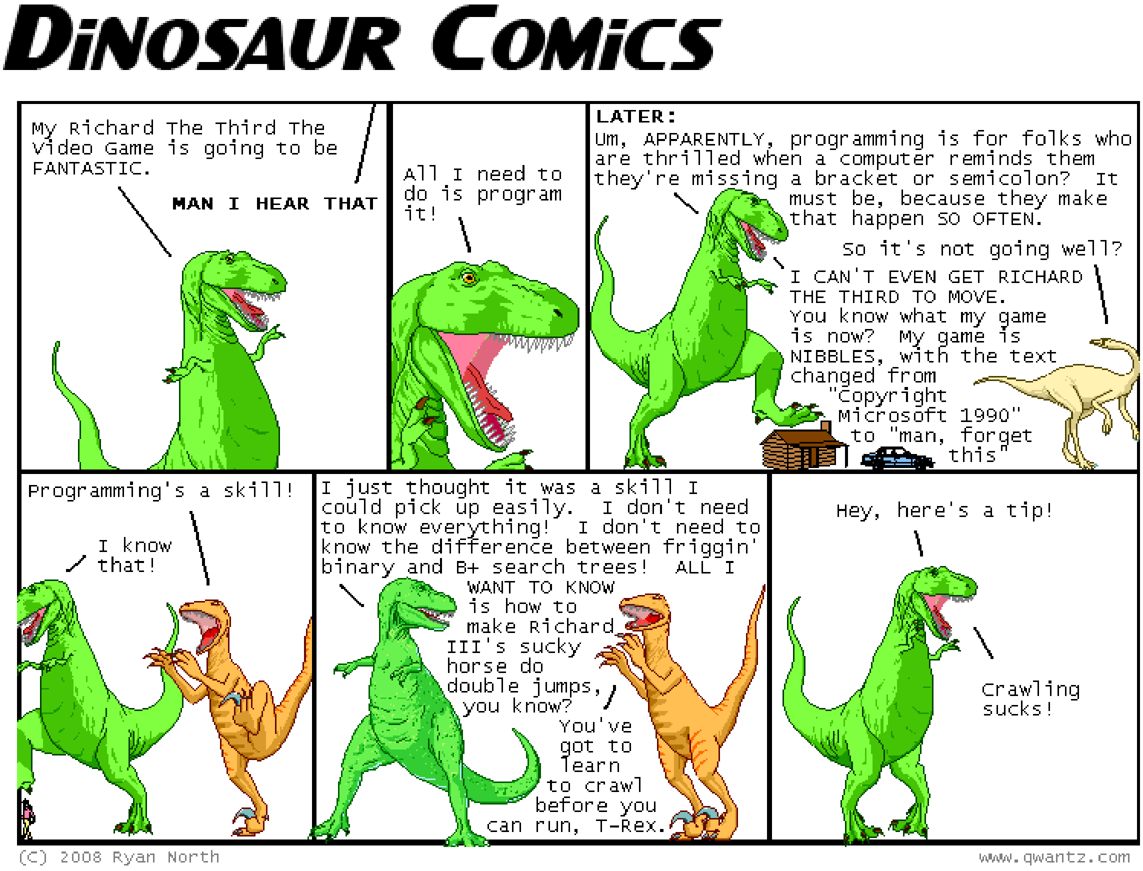
I ve also made this book available under the Creative Commons license, which allows you to make copies and distribute this book (or excerpts) with my full permission, as long as attribution to me is left intact and it is used for noncommercial purposes. (See the copyright page.) I want to make this book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

What's New in the 3rd Edition?

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

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

The source code has intentionally been kept the same as the second edition to prevent confusion. If y " " new to programming, or introducing a friend to programming, this third edition will make the process easier, smoother, and more fun.


DINOSAUR COMICS
My Richard The Third The Video Game is going to be FANTASTIC.
MAN I HEAR THAT
All I need to do is program it!
LATER:
Um, APPARENTLY, programming is for folks who are thrilled when a computer reminds them they're missing a bracket or semicolon? It must be, because they make that happen SO OFTEN.
So it's not going well?
I CAN'T EVEN GET RICHARD THE THIRD TO MOVE. You know what my game is now? My game is NIBBLES, with the text changed from "Copyright Microsoft 1990" to "man, forget this"
Programming's a skill!
I know that!
I just thought it was a skill I could pick up easily. I don't need to know everything! I don't need to know the difference between friggin' binary and B+ search trees! ALL I WANT TO KNOW is how to make Richard III's sucky horse do double jumps, you know?
You've got to learn to crawl before you can run, T-Rex.
Hey, here's a tip!
Crawling sucks!
(C) 2008 Ryan North
www.qwantz.com

Who is this book for?

Programming hard. But it is hard to find learning materials that teach you to do interesting things with programming. Other computer books go over many topics most newbie coders need. This book will teach you how to program your own computer games. learn a useful skill and have fun games to show for it! This book is for:

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

Topics Covered In This Chapter:

- Downloading and installing the Python interpreter
- How to use this book
- http://inventwithpython.com

Hello! This book teaches you how to program by making video games. Once you learn how the
" "
some software called the Python interpreter, and this book. The Python interpreter is free to download from the Internet.

When I was a kid, a book like this one taught me how to write my first programs and games. It was fun and easy. Now as an adult, I still have fun programming and I get paid for it. But even if
"
skill to have.

Computers are incredible machines, and learning to program them is . If you can read this book, you can program a computer. A computer **program** is a bunch of instructions that the computer can understand, just like a storybook is a bunch of sentences understood by the reader. Since video games are nothing but computer programs, they are also made up of instructions.

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

When I was a kid, BASIC was a common first language to learn. However, new programming languages such as Python have been invented since then. Python is even easier to learn than

adults use Python in their work and when programming for fun.

"
PlayStation, or Nintendo
G
have to be complicated to be fun.

Downloading and Installing Python

d to install software called the Python interpreter. The **interpreter** program

Important Note! Be sure to install Python 3, and not Python 2. The programs in
" th Python 2.
It is so important -1 to tell you to install
Python 3 so you do not miss this message.

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

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

1. Select Install for All Users and then click **Next**.
2. Install to the *C:\Python34* folder by clicking **Next**.
3. Click **Next** to skip the Customize Python section.

On Mac OS X, download the *.dmg*
double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. When the DMG package opens in a new window, double-click the *Python.mpkg* file. You may have to enter the administrator password.
2. Click **Continue** through the Welcome section and click **Agree** to accept the license.

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

"

these steps:

1. Open the Ubuntu Software Center.
2. Type *Python* in the search box in the top-right corner of the window.
3. Select **IDLE (using Python 3.4)**, or whatever is the latest version.
4. Click **Install**. You may have to enter the administrator password to complete the installation.

Starting IDLE

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

On Windows, click the Start button in the lower " **IDLE (Python GUI)**.

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

"

click on **Applications** at the top of the screen. Then click **Programming** and **IDLE 3**.

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

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

How to Use this Book

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

Type the code for the program into file editor yourself, rather than download or copy/paste take the time to type in the code.

Line Numbers and Spaces

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

"

type it like this:

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

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

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

```
▪▪▪▪
▪▪▪▪▪▪▪▪
```

Text Wrapping in This Book

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

The first instruction wraps around and makes it look like three s only instruction on one line.

Finding Help Online

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

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

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

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

- If you are typing out the programs in this book but getting an error, first check for typos with the online diff tool at http://invpy.com/diff. Copy and paste your code into the diff tool
- Explain what you are trying to do when you explain the error. This will let your helper know if you are on the wrong path entirely.
- Copy and paste the entire error message and your code.
- Search the Web to see whether someone else has already asked (and answered) your question.
- already put in some work to try to figure things out on your own.
- emand help or pressure your helpers to respond quickly.

"

you are trying to do, the exact error you are getting, and your operating system and version.

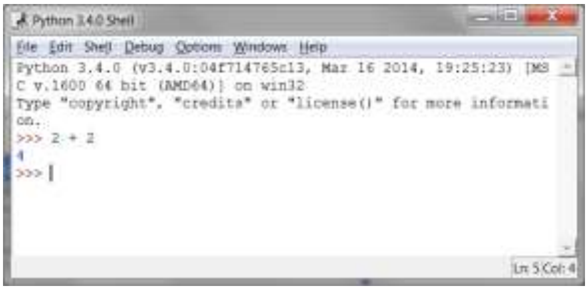
Topics Covered In This Chapter:

- Integers and Floating Point Numbers
- Expressions
- Values
- Operators
- Evaluating Expressions
- Storing Values in Variables

Before you can make games, you need to learn a few basic programming concepts. You make games in this chapter, but learning these concepts is the first step to programming video games.

Some Simple Math Stuff

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



```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MS
C v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more informati
on.
>>> 2 + 2
4
>>> 
```

Figure 2-1: Enter 2+2 into the interactive shell.

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

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

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division

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

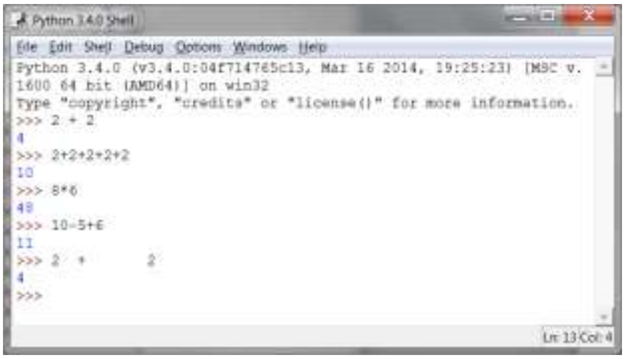
Integers and Floating Point Numbers

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

Expressions

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

After you type in the above instructions, the interactive shell will look like Figure 2-2.



```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.
1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 + 2
4
>>> 2+2+2+2+2
10
>>> 8*6
48
>>> 10-5+6
11
>>> 2  +       2
4
>>>
```

Figure 2-2: What the IDLE window looks like after entering instructions.

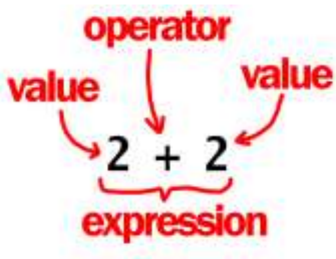


Figure 2-3: An expression is a made up of values and operators.

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

Evaluating Expressions

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

Expressions can be of any size, but they will always evaluate down to a single value. Even single values are expressions: The expression evaluates to the value . For example, the expression will evaluate down to the value through the following steps:

```
                    ▼

                    ▼

                    ▼

                    ▼

                    ▼
```

shows you the results:

Notice that the division operator evaluates to a float value, as in evaluating to . Math operations with float values also evaluate to float values, as in evaluating to .

Syntax Errors

If you enter into "

This error happened because operators. But the + operator expects a value after the sign. An error message appears when this value is missing.

"

to tell it.

instruction correctly into the interactive shell at the next prompt.

Storing Values in Variables

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

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

The stored in it, as shown in Figure 2-4. The name

small note inside the box.

When you press ENTER you see anything in response. In Python, the instruction executed was successful if no error message appears. The prompt will appear so you can type in the next instruction.

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

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

Variables store values, not expressions. For example, consider the expression in the statements and . They both evaluate to . The end result is the same: Both assignment statements store the value in the variable .

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

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

to , so writing is like writing the expression . Here are the steps of being evaluated:

```
      ▼

      ▼
```

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

The error appeared because variable but no variable named .

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

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

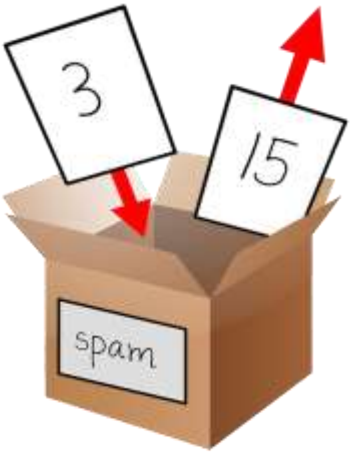


Figure 2-5: The 15 value in spam being overwritten by the 3 value.

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

The assignment statement " variable will be the current value of Keep increasing the value in by several times by entering the following into the interactive shell:

Using More Than One Variable

Create as many variables as you need in your programs " to two variables named and , like so:

Now the variable has inside it, and has inside it. Each variable is its own box with its own value, like in Figure 2-6.

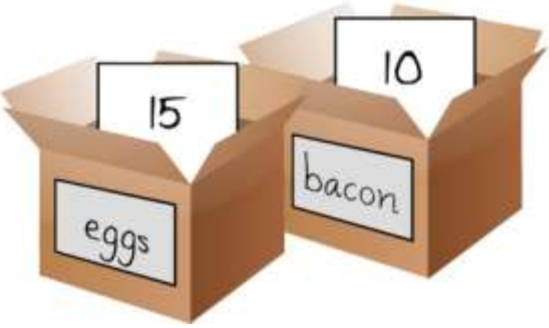


Figure 2-6: The “bacon” and “eggs” variables have values stored in them.

Try entering into the interactive shell, then check the new value of :

The value in is now . When you added and you are adding their values, which are and , respectively. Variables contain values, not expressions. The variable was assigned value , and not the expression . After the assignment statement, changing or does not affect .

Summary

In this chapter, you learned the basics about writing Python instructions. Python needs you to tell y understand specific instructions.

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

There are many other types of operators and values in Python. In the next chapter, ll go over some more basic concepts and write your first program. ll learn about working with text in expressions. Pytho

Topics Covered In This Chapter:

- Flow of execution
- Strings
- String concatenation
- Data types (such as strings or integers)
- Using the file editor to write progams
- Saving and running programs in IDLE
- The function
- The function
- Comments
- Case-sensitivity

"

learn how to store text in variables, combine text, and display text on the screen.

Almost all programs display text to the user, and the user enters text into your programs through in this chapter. This program displays the " name.

Strings

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

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

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

String Concatenation

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

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

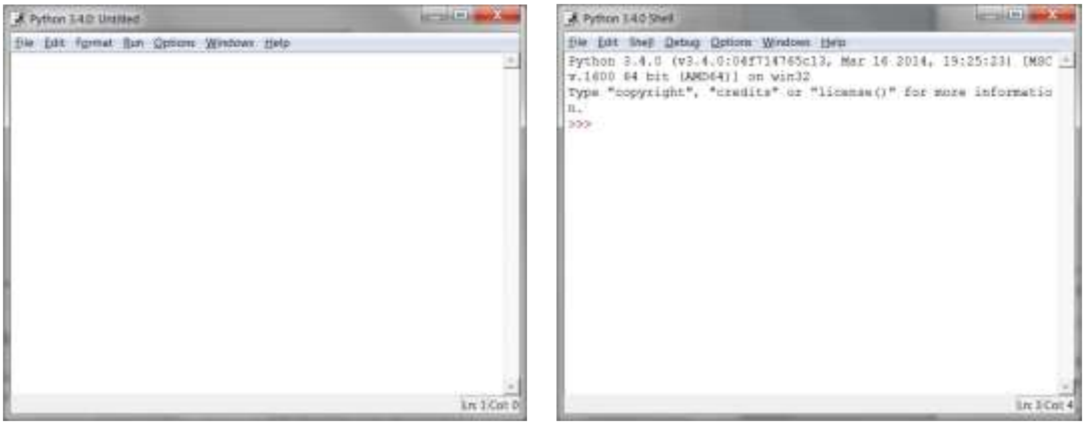
The operator works differently on string and integer values because they are different data types. All values have a data type. The data type of the value is a string. The data type of the value is an integer. The data type tells Python what operators should do when evaluating expressions. The operator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

Writing Programs in IDLE's File Editor

" ions into interactive shell one at a time. When you "

your first program!

IDLE has another part called the **file editor**. Click on the **File** menu at the top of the interactive shell window. Then select **New Window**. A blank window will appear for you to type your , like in Figure 3-1.


Python 3.4.0: Untitled
File Edit Format Run Options Windows Help
Ln: 1 Col: 0
Python 3.4.0 Shell
File Edit Shell Debug Options Windows Help
Ln: 3 Col: 4

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC
v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more informatio
n.
>>>
```

Figure 3-1: The file editor window (left) and the interactive shell window (right).

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

Hello World!

own Hello World program now.

"

so this book can refer to code by line number. The bottom-right corner of the file editor window will tell you where the blinking cursor is. Figure 3-2 shows that the cursor is on line 1 and column 0.

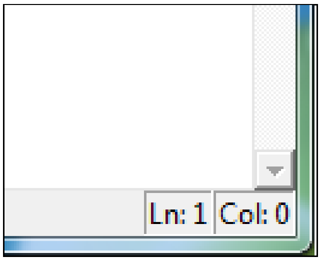


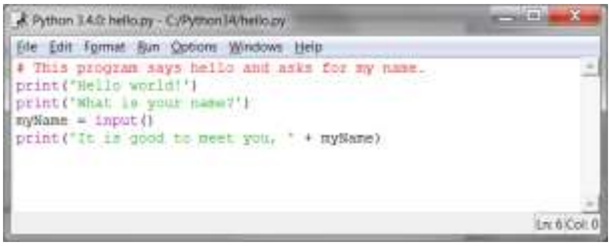
Figure 3-2: The bottom right of the file editor window tells you what line the cursor is on.

hello.py

source code. It contains the instructions Python will follow when the program is run.

IMPORTANT NOTE! The programs in this book will only run on Python 3, not " Python 3.4.2 ou have Python 2 installed, you can have Python 3 installed at the same time. To download Python 3, go to https://python.org/download/.

done typing the code, the window should look like this:

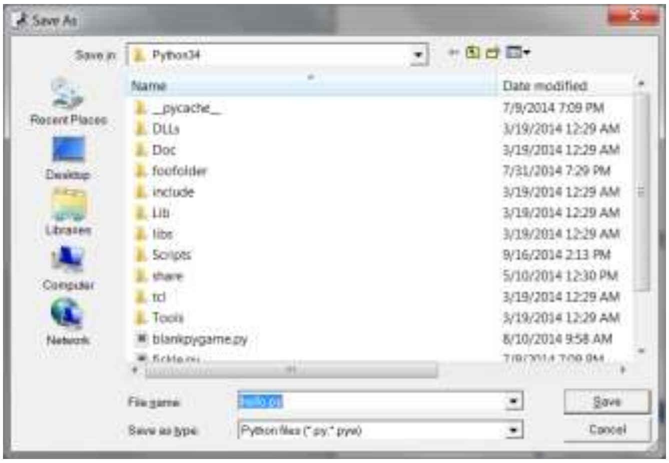

Python 3.4.0: hello.py - C:/Python34/hello.py
File Edit Format Run Options Windows Help
Ln: 6 Col: 0

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

Figure 3-3: The file editor window will look like this after you type in the code.

Saving Your Program

ource code, save it by clicking on **File ► Save As**. Or press Ctrl-S to save with a keyboard shortcut. Figure 3-4 shows the Save As window that will open. Enter *hello.py* in the **File name** text field then click **Save**.



Name	Date modified
__pycache__	7/9/2014 7:09 PM
DLLs	3/19/2014 12:29 AM
Doc	3/19/2014 12:29 AM
foofolder	7/31/2014 7:29 PM
include	3/19/2014 12:29 AM
Lib	3/19/2014 12:29 AM
libs	3/19/2014 12:29 AM
Scripts	9/16/2014 2:13 PM
share	5/10/2014 12:30 PM
tcl	3/19/2014 12:29 AM
Tools	3/19/2014 12:29 AM
blankpygame.py	8/10/2014 9:58 AM
fickle.py	7/9/2014 7:09 PM

File name: hello.py

Save as type: Python files (*.py,*.pyw)

Figure 3-4: Saving the program.

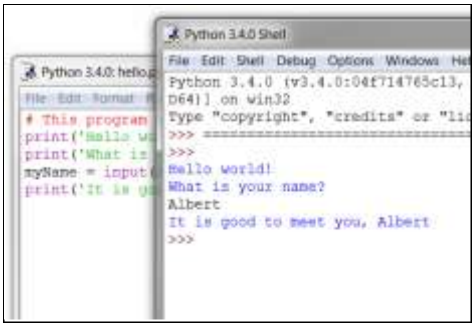
You should save your programs often while you type them. That way, if the computer crashes or you accidentally exit from IDLE you lose much work.

Opening The Programs You've Saved

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

File ► Run ► Run Module or just press F5 from the file editor window. Your program will run in the interactive shell window.

Enter your name when the program asks for it. This will look like Figure 3-5.



```
# This program
print('Hello w
print('What is
myName = input(
print('It is g
```

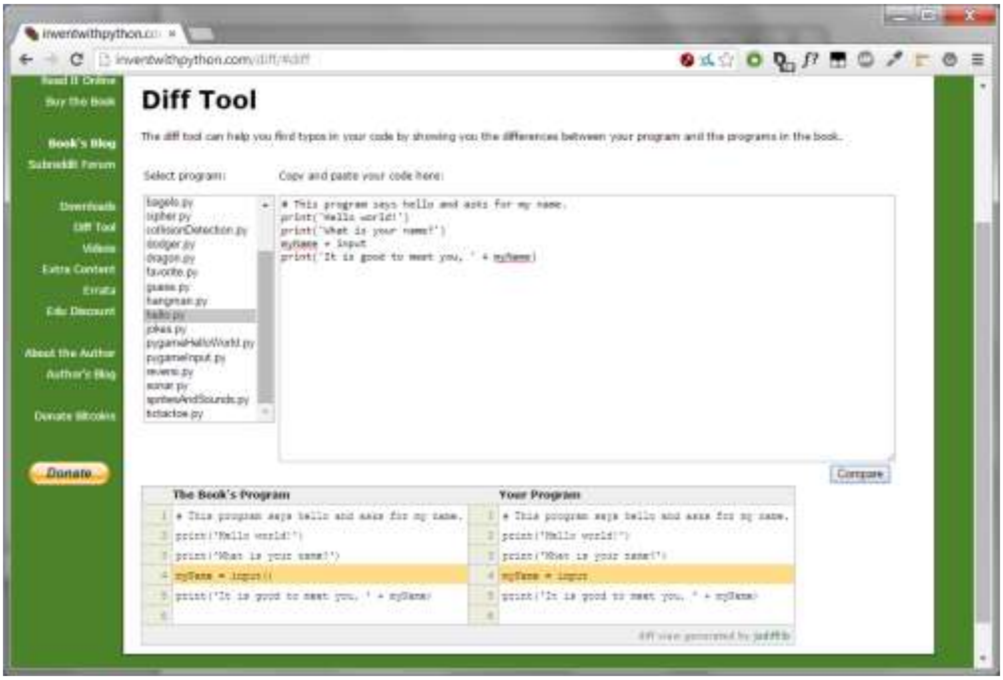
```
Python 3.4.0 (v3.4.0:04f714765c13,
D64)] on win32
Type "copyright", "credits" or "lic
>>> ================================
>>>
Hello world!
What is your name?
Albert
It is good to meet you, Albert
>>>
```

Figure 3-5: The interactive shell after running hello.py.

When you type your name and push ENTER, the program will greet you by name.

F5 again to run the program a second time and enter another name.

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



Diff Tool

The diff tool can help you find typos in your code by showing you the differences between your program and the programs in the book.

Select program:

Copy and paste your code here:

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

The Book's Program

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

Your Program

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

diff view generated by jsdifflib

Figure 3-6: The diff tool at http://invpy.com/diff

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

```
Albert
```

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

How the “Hello World” Program Works

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

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

ll begin with line number 1.

Comments

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

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

Functions

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

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

To display on the screen, type the function name, followed by an opening parenthesis, followed by the string and a closing parenthesis.

The `print()` function

Lines 2 and 3 are calls to the function. A value between the parentheses in a function call is an **argument** function call is . The argume function call is . This is called **passing** the argument to the function.

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

The `input()` function

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

The value that the function call evaluates to is called the **return value**. (In fact, the value a function call s the value a function call evaluates to case, the return value of the function is the string that the user typed in-their name. If the " function call evaluates to the string . The evaluation looks like this:

▼

This is how the string value gets stored in the variable.

Using Expressions in Function Calls

The last line is another function call. The expression in between the parentheses of . However, arguments are always single values. Python will first evaluate this expression and then pass that value as the argument. If is stored in , the evaluation looks like this:

```
▼

▼
```

This is how the program greets the user by name.

Ending the Program

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

```
Carolyn
```

Remember, the computer does exactly what you program it to do. Computers are dumb and just follow the instructions you give it exactly you type in your name, someone name, or just something silly. Type in anything you want. The computer will treat it the same way:

```
poop
```

Variable Names

Giving variables descriptive names makes it easier to understand what a program does. Imagine if

helpful at all!

Instead of , you could have called this variable or . Python do care. It will run the program just the same.

Variable names are case-sensitive. **Case-sensitive** means the same variable name in a different case is considered a different variable. So , , , and are four different variables in Python. They each contain their own separate values.
variables in your program. Use descriptive names for your variables instead.

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

Short variable names are better than long names: or is more readable than .

, , , and . This
"

use descriptive names. Your programs should use descriptive variable names too.

Summary

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

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

Functions are used to carry out some complicated instruction as part of your program. Python has many built- Function calls can be used in expressions anywhere a value is used.

The instruction in your program that Python is currently at is called the execution. In the next
"
the program. On "

Topics Covered In This Chapter:

- statements
- Modules
- statements
- Conditions
- Blocks
- Booleans
- Comparison operators
- The difference between and
- statements
- The keyword
- The and and functions
- The function

"

random number from 1 to 20, and ask you to guess it. The computer will tell you if each guess is too high or too low. You win if you can guess the number within six tries.

This is a good game to code because it uses random numbers, loops, and input from the user in a data types, and why you would need to do this. Since this program is a game, call the user the **player**. B correct too.

Sample Run of Guess the Number

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

```

Albert


10


2
```

4

Source Code of Guess the Number

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

> **IMPORTANT NOTE!** The programs in this book will only run on Python 3, not
> " Python
> 3.4.2 "
> at the same time. To download Python 3, go to https://python.org/download/.

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

import statements

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

The second line is an **import statement**. Remember, statements are instructions that perform

assignment statements store a value in a variable.

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

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

Line 4 creates a new variable named at this point in the program, store the integer here.

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

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

Line 7 lets the user type in their name and stores it in the variable. (Remember, the string

dumb and just follow their instructions no matter what.)

The random.randint() Function

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

The function is provided by the module, so you must precede it with is in the module.

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

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

Use the function when you want to add randomness to your games. Y
randomness in many games. (Think of how many board games use dice.)

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

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

9 and 10 from this:

lines:

```
                                  100
                                                                 100
```

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

Welcoming the Player

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

"

The plus signs concatenate the three strings to evaluate down to one string. And that one string is the argument passed to the function "
inside the quotes and part of the strings themselves.

Loops

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

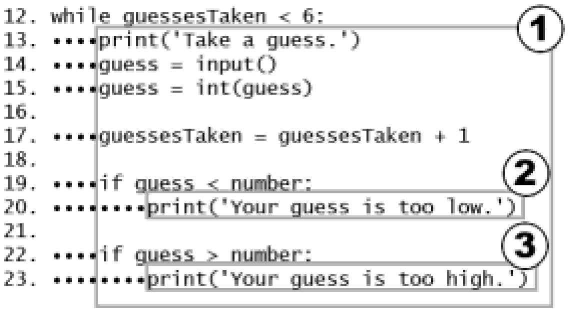
Blocks

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

also indented by four spaces is part of the block. The block ends when a line of code with the same indentation before the block started. This means blocks can exist within other blocks. Figure 4-1 is a diagram of code with the blocks outlined and numbered.

In Figure 4- "
four spaces. Si "
started. This block is labeled (1) in Figure 4-1. This block will continue until a line with zero spaces (the original indentation before the block began). Blank lines are ignored.

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



```
12. while guessesTaken < 6:
13. ••••print('Take a guess.')
14. ••••guess = input()
15. ••••guess = int(guess)
16.
17. ••••guessesTaken = guessesTaken + 1
18.
19. ••••if guess < number:
20. ••••••••print('Your guess is too low.')
21.
22. ••••if guess > number:
23. ••••••••print('Your guess is too high.')
```

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

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

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

" ines 13 to 23 all in one block marked (1). Line 20 is in a block in a block marked as (2). Line 23 is the only line in another block in a block marked as (3).

The Boolean Data Type

The Boolean data type has only two values: or . These values must be typed with a

values (called **bools** for short) with comparison operators to form conditions. (Conditions are explained later.)

For example, try storing the Boolean values in variables:

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

Comparison Operators

Line 12 has a statement:

The expression that follows the keyword (the part) contains two values (the value in the variable , and the integer value) connected by an operator (the sign is a **comparison operator**.

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

Table 4-1: Comparison operators.

Operator Sign	Operator Name
	Less than
	Greater than
	Less than or equal to
	Greater than or equal to
	Equal to
	Not equal to

! "-, *, and / math operators. Like any operator, the comparison operators combine with values to form expressions such as .

Conditions

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

For example, the condition " less than the number " . If not, the condition evaluates to .

" in . Because is less than , this condition evaluates to the Boolean value of . The evaluation would look like this:

```
            ▼

            ▼
```

Experiment with Booleans, Comparison Operators, and Conditions

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

The condition returns the Boolean value because the number is less than the number . But because , the condition evaluates to .
than , so is . is less than , so is .

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

Now try entering these expressions into the interactive shell:

The Difference Between = and ==

Try not to confuse the assignment operator ( ). The equal sign ( ) is used in assignment statements to store a value to a variable, while the equal-equal sign ( ) is used in expre
accidentally use one when you meant to use the other.

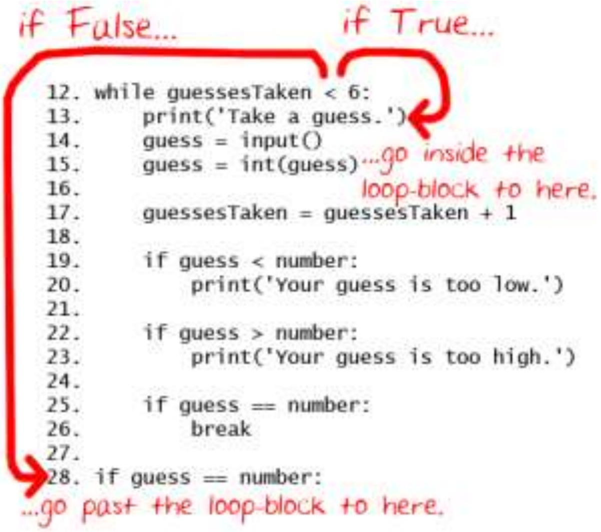
) has two characters in it, just as the
) has two characters in it.

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

Looping with `while` statements

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

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



```
12. while guessesTaken < 6:
13.     print('Take a guess.')
14.     guess = input()
15.     guess = int(guess)
16.
17.     guessesTaken = guessesTaken + 1
18.
19.     if guess < number:
20.         print('Your guess is too low.')
21.
22.     if guess > number:
23.         print('Your guess is too high.')
24.
25.     if guess == number:
26.         break
27.
28. if guess == number:
```

if False...
if True...
...go inside the loop-block to here.
...go past the loop-block to here.

Figure 4-2: The loop's condition.

Figure 4-2 shows how the execution flows depending on the condition. If the condition evaluates to (which it does the first time, because the value of is), execution will enter the while-block at line 13 and keep going down. Once the program reaches the end of the while-block, instead of going down to the next line, the execution loops back up to the -evaluates the condition. As before, if the condition is the execution enters the while-block again. Each time the execution goes through the loop is called an **iteration**.

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

The Player Guesses

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

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

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

The call will return the integer value . However, even though you can pass a string to the function, you cannot pass just any string. Passing to will result in an error. The string you pass to must be made up of numbers:

The line shows an expression that uses the return value of as part of an expression. It evaluates to the integer value :

```
▼

▼
```

Remember, the function always returns **a string** of text the player typed. If the player types , the function will return the string value , not the integer value . Python cannot use the and comparison operators to compare a string and an integer value:

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

One last thing: Calling variable. The code is an expression that evaluates to the integer value form of the string stored in the variable. What changes is the assignment statement:

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

Using the , , , and functions, you can take a value of one data type and return it as a value of a different data type.

Incrementing Variables

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

On the first iteration of the loop, has the value of . Python will take this value and add to it. evaluates to , which is stored as the new value of . Think of "

Adding one to a v called **incrementing** the variable. Subtracting called **decrementing** the variable.

if statements

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

"

execution moves inside the if-block on line 20 and prints a message telling the player this.

The statement works almost the same as a statement, too. But unlike the while-block, statement at the end of the if-block. It just continues down to the next line. In other words, statements See Figure 4-3 for a comparison of the two statements.

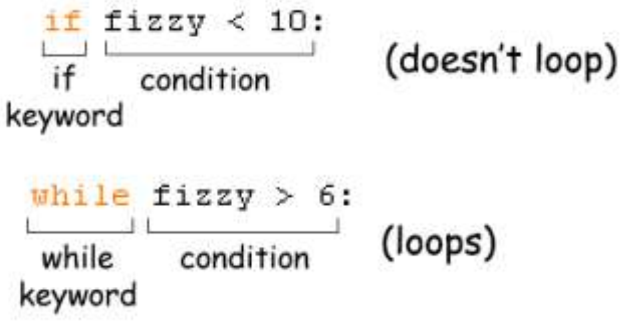


Figure 4-3: if and statements.

secret number. If this condition is , then the function call tells the player that their guess is too high.

Leaving Loops Early with the break statement

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

A **break statement** tells the execution to jump immediately out of the while-block to the first line after the end of the while-block. The statement condition.

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

secret number, the execution reaches the bottom of the while-block. This means the execution will loop back to the top and recheck the condition on line 12 ( ). Remember after the instruction executed, the new value of is . Because is , the execution enters the loop again.

If the player keeps guessing too low or too high, the value of will change to , then , then , then , then . When has the number stored in it, the

( ) is , since s than . Because the , the execution moves to the first line after the while-block, line 28.

Check if the Player Won

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

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

Lines 29 and 30 only execute if the condition in the statement on line 28 was (that is, if

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

Check if the Player Lost

Line 32 uses the comparison operator al to the secret number. If this condition evaluates to , the execution moves into the if-block on line 33.

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

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

At this point, the execution has reached the end of the code, and the program terminates. rst real game!

the player only four guesses, change the code on line 12:

into this line:

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

Flow Control Statements

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

Summary

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

"But what exactly is a program?" When you see someone using a computer program (for
" "
screen. The prog **output**), based

input). A

program

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

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

" " ails about those four types of "

statements, and many other functions that come with Python. There are also different types of I/O such as input from the mouse or outputting sound and graphics instead of just text.

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

Topics Covered In This Chapter:

- Escape characters
- Using single quotes and double quotes for strings
- Using keyword argument to skip newlines

Making the Most of `print()`

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

ogram tells a few different jokes to the user, and demonstrates advanced string and code.

Sample Run of Jokes

Source Code of Jokes

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

IMPORTANT NOTE! The programs in this book will only run on Python 3, not " Python 3.4.2 " 3 installed at the same time. To download Python 3, go to https://python.org/download/.

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

How the Code Works

Lines 1 to 4 have three function
joke " function after the first . The player can read the joke, press ENTER, and then read the punch line.

The user can still type in a string and hit ENTER"
variable. The program will just forget about it and move to the next line of code.

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

Escape Characters

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

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

Some Other Escape Characters

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

The escape character t simulates pushing the Tab key on your keyboard. Instead, try this line:

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

Table 5-1: Escape Characters

Escape Character	What Is Actually Printed
	Backslash (\)
	Single quote (')
	Double quote (")
	Newline
	Tab

Quotes and Double Quotes

double quotes. These two lines print the same thing:

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

I like to use single
"

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

"
: . The Python interpreter is smart enough
"
string is ending.

print()'s end Keyword Argument

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

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

By passing a blank string for , the function add a newline at the end of the string, but instead add a blank string. This is why appears next to the previous line, instead of on its own new line. There was no newline after the string was printed.

Summary

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

The function automatically appends a newline character to the end of the string passed it to be displayed on the screen. Most of the time, this is a helpful shortcut. But sometimes you
" keyword
"
character, you would call .

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

Topics Covered In This Chapter:

- The function
- Creating your own functions with the keyword
- The keyword
- The , , and Boolean operators
- Truth tables
- Global and local variable scope
- Parameters and Arguments
- Flow charts

Functions

, , , , and .
"

own functions for your programs to call. A function is like a mini-program inside a program.

Functions let you run the same code multiple times without duplicating the source code several times. Instead, you can put that code inside a function and call the function several times. This
"

program to change it.

The game you will create in this chapter i
two caves which hold either treasure or certain doom.

How to Play Dragon Realm

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

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

Sample Run of Dragon Realm

```
1

no
```

Source Code of Dragon Realm

IMPORTANT NOTE! The programs in this book will only run on Python 3, not
" Python
3.4.2 "
at the same time. To download Python 3, go to https://python.org/download/.

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

How the Code Works

This program imports two modules. The module will provide the -related functions that the module includes, so line 2 imports the module.

`def` Statements

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

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

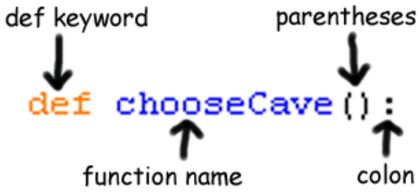


Figure 6-1: Parts of a def statement.

Remember, the
when you call the function. When the execution reaches a statement it skips down to the first line after the def-block.

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

Then all of the
displayed.

Where to Put Function Definitions

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

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

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

Defining the chooseCave() Function

Line 11 defines another function called
which cave they want to go in, either 1 or 2.

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

Line 12 creates a new variable called and stores a blank string in it. T

" is true and " **and** true.

" . , dogs do not have wing " " " only true if **both** parts are true. If one or both parts are false, then the entire sentence is false.

The and and or Operators

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

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

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

Try entering the following into the interactive shell:

The not Operator

The operator only works on one value, instead of combining two values. The operator evaluates to the opposite Boolean value. The expression will evaluate to and will evaluate to .

Try entering the following into the interactive shell:

Truth Tables

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

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

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

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

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

Table 6-3: The operator's truth table.

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

Evaluating Boolean Operators

Look at line 13 again:

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

The first time the " is set to the blank string, . The blank string is not equal to the string , so the left side evaluates to . The blank string is also not equal to the string , so the right side evaluates to .

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

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

```
▼

▼

▼

▼

▼
```

Getting the Player's Input

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

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

▼

▼

▼

▼

▼

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

Return Values

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

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

Skip down and look at line 40 for a moment:

When the function is later called by the program on line 40, the return value is stored in the variable. The loop guarantees that will only return either or as its return value.

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

Global Scope and Local Scope

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

When execution is inside a function, you cannot change the variables outside of the function, including variables inside other functions. This is because these variables exist in a different

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

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

You can read the value of global variables from the local scope, but attempting to change a global variable from the local scope work. What Python actually does in that case is create a local variable with the **same name** as the global variable. You could, for example, have a local variable named at the same time as having a global variable named . Python will consider these to be two different variables.

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

When run, this code will output the following:

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

...the variable is created inside the function. This means it is created in the returns, and will be recreated if remembered between function calls.

Parameters

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

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

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

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

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

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

Displaying the Game Results

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

Here the code prints some more text and waits for another 2 seconds. These short pauses add

"

program, you called the function to pause until the player pressed the ENTER key. Here, to do anything except wait a couple seconds.

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

Deciding Which Cave has the Friendly Dragon

Line 28 calls the function which will return either or This integer value is stored in and is the cave with the friendly dragon.

variable (or) is equal to the friendly dragon cave.

But the value in is an integer because returns integers. You sign, because they will **always** be not equal to each other. is not equal to and is not equal to .

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

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

Line 32 is an **else** statement. The statement can only come after an if-block. The else-block executes if the statement . "
-block or else execute the else-

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

Where the Main Part Begins

statement or inside a def-block. This line is where the main part of the program begins. The previous statements merely defined the functions. They

Line 35 and 36 are setting up a loop that the rest of the game code is in. At the end of the game, the player can enter if they want to play again. If they do, the execution enters the loop to "
and the execution will move on to the end of the program and terminate.

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

Calling the Functions in the Program

Line 38 calls the "

you defined earlier on line 4. When this function is called, the program execution jumps to the first line in the function on line 5. When all the lines in the function are done, the execution jumps back to line 38 and continues moving down.

Line 40 also calls a function that you defined. Remember that the function lets the player type in the cave they want executes, the program execution jumps back to line 40, and the call evaluates to the return value. This return value is stored in a new variable named . Then the program execution moves on to line 42.

Line 42 calls your function, passing the value in as an argument. Not only does execution jump to line 20, but the value in is copied to the parameter inside the function. This is the function that will display either or depending on the cave the player chose to go into.

Asking the Player to Play Again

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

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

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

One thing to note: the string is not equal to the string . If the player typed in the string , then the and the program would still terminate. Later programs in this book will show you how to avoid this problem.

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

In the next chapter you create a game, but instead learn how to use a feature of IDLE called the debugger.

Designing the Program

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

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

" row from the box to another box. Your finger is like the program execution. The program terminates when

"

"

Summary

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

on is called. The function call itself evaluates to the return value.

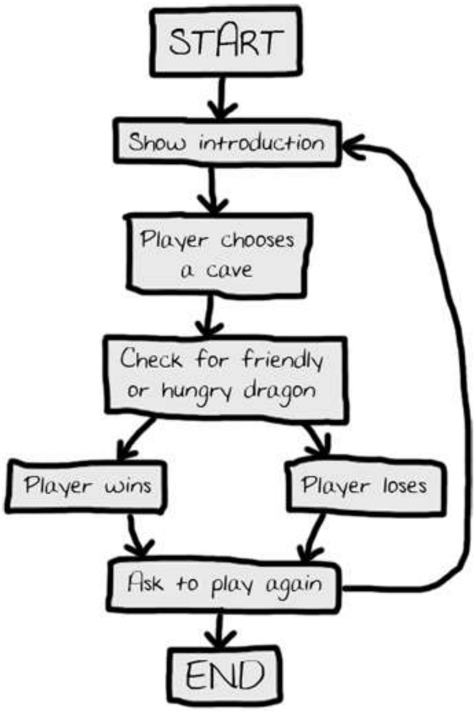


Figure 6-2: Flow chart for the Dragon Realm game.

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

Variable scopes might seem complicated, but they are useful for organizing functions as separate pieces of code from the rest of the program. Because each function has its own local scope, you can be sure that the code in one function cause bugs in other functions.

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

Topics Covered In This Chapter:

- 3 Different Types of Errors
-
- Stepping Into, Over, and Out
- Go and Quit
- Break Points

Bugs!

asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion

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

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

- **Syntax Errors** are a type of bug that comes from typos. When the Python interpreter sees
  "
  program with even a single syntax error run.
- **Runtime Errors** are bugs that happen while the program is running. The program will work up until it reaches the line of code with the error, and then the program terminates with an error message (this is called **crashing**). The Python interpreter will display a
  ppens.
- **Semantic Errors** "
  doing what the programmer intended for the program to do. For example, if the programmer wants the variable to be the *sum* of the values in variables , , and but writes , then the value in total will be wrong. This could crash the program later on, but it is not immediately obvious where the semantic bug happened.

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

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

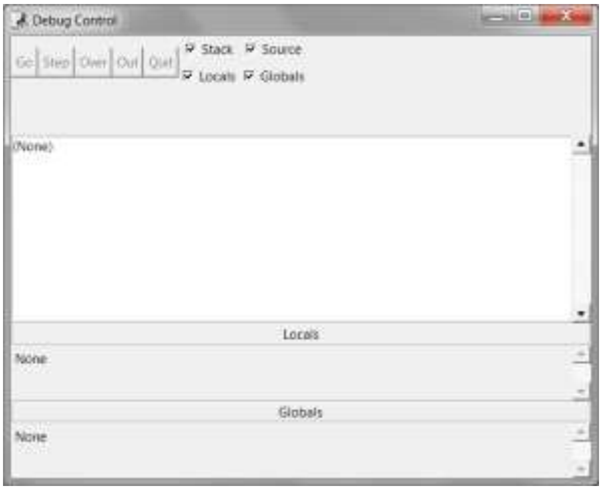
Now press and hold down the Ctrl key and press the C key to stop the program. The interactive shell will look like this:

The Debugger

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

Starting the Debugger

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

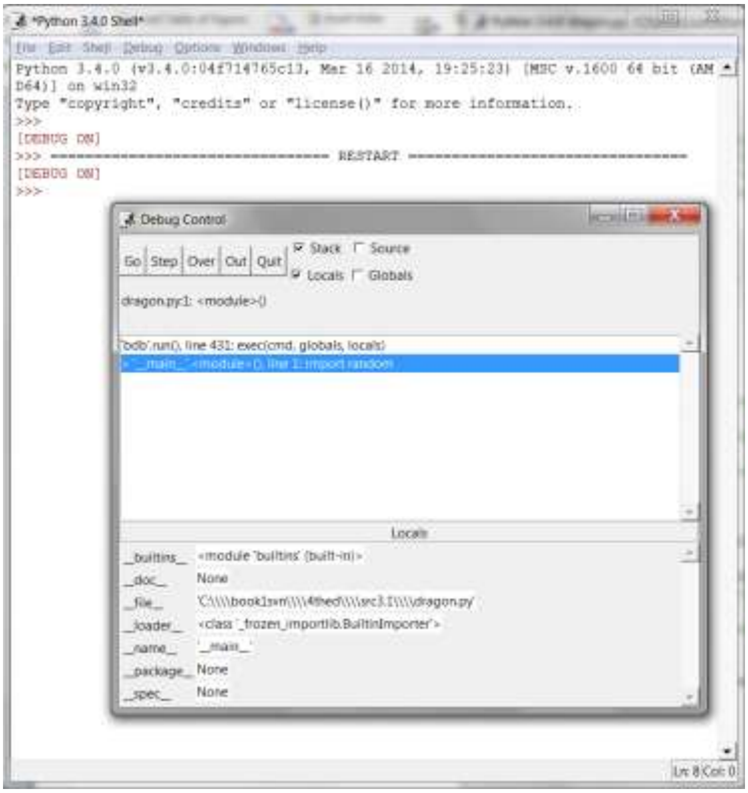


```
(None)
```

Locals: None

Globals: None

Figure 7-1: The Debug Control window.



```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
[DEBUG ON]
>>> ================================ RESTART ================================
[DEBUG ON]
>>>
```

```
dragon.py:1: <module>()

'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__'.<module>(), line 1: import random
```

```
__builtins__   <module 'builtins' (built-in)>
__doc__        None
__file__       'C:\\\\book1svn\\\\4thed\\\\src3.1\\\\dragon.py'
__loader__     <class '_frozen_importlib.BuiltinImporter'>
__name__       '__main__'
__package__    None
__spec__       None
```

Figure 7-2: Running the Dragon Realm game under the debugger.

Now when you run the Dragon Realm game by pressing **F5**"
" **Source**
and **Globals** checkboxes.

When you run Python programs under the debugger, the program will stop before it executes the first instruction □ **Source** checkbox in the Debug Control window), the first instruction is highlighted in gray. The Debug Control window shows the execution is on line 1, which is the line.

Stepping

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

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

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

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

Globals Area

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

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

"

module 'random' from 'C:\\Python31\\lib\\ □ Y means to debug your programs. Just seeing that the functions and modules are there in the Global area will tell you if the function has been defined or the module has been imported.

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

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

Locals Area

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

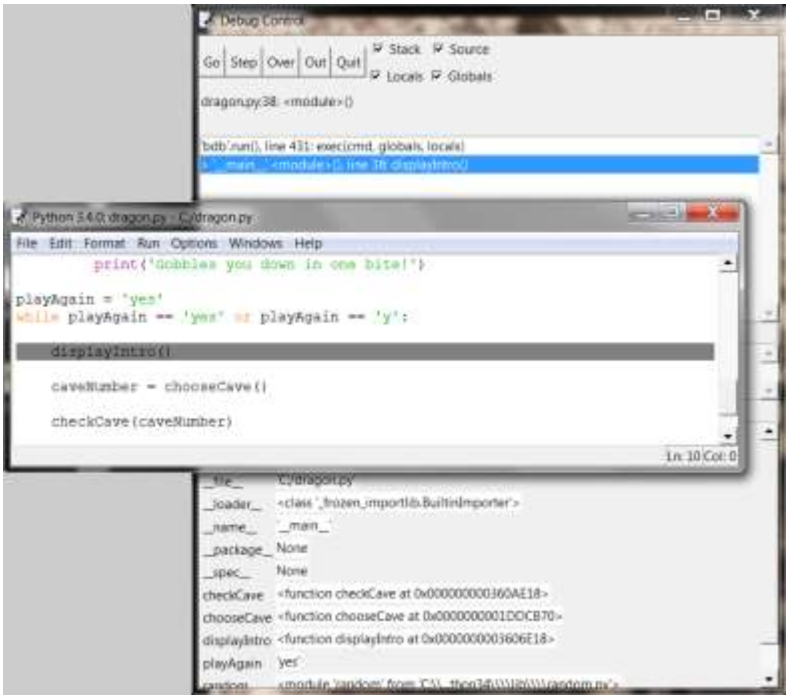
The Go and Quit Buttons

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

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

Stepping Into, Over, and Out

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



dragon.py:38: <module>()

'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__'.<module>(), line 38: displayIntro()

```
        print('Gobbles you down in one bite!')

playAgain = 'yes'
while playAgain == 'yes' or playAgain == 'y':

    displayIntro()

    caveNumber = chooseCave()

    checkCave(caveNumber)
```

__file__	'C:/dragon.py'
__loader__	<class '_frozen_importlib.BuiltinImporter'>
__name__	'__main__'
__package__	None
__spec__	None
checkCave	<function checkCave at 0x000000000360AE18>
chooseCave	<function chooseCave at 0x0000000001DDCB70>
displayIntro	<function displayIntro at 0x0000000003606E18>
playAgain	'yes'
random	<module 'random' from 'C:\\...thon34\\\\lib\\\\random.py'>

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

When the execution is paused at line 5, clicking **Step** one more time will step into the function. The built- " , , , or

causing bugs in your program.

function. So instead of clicking **Step** to step into the " **Over**. This will step *over* the code inside the function. The code inside will be executed at normal speed, and then the debugger will pause once the execution returns from .

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

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

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

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

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

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

a recap of what each button does:

- **Go -** Executes the rest of the code as normal, or until it reaches a break point. (Break points are described later.)
- **Step -** Step one instruction. If the line is a function call, the debugger will step into the function.
- **Over -** Step one instruction. If the line is a function call, the debugger *step into* the function, but instead *step over* the call.
- **Out -** Keeps stepping over lines of code until the debugger leaves the function it was in when **Out** was clicked. This *steps out* of the function.
- **Quit -** Immediately terminates the program.

Find the Bug

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

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

```
6
```

t crash but it is not working correctly. The program says the user is wrong even if they type the correct answer.

interactive shell window, click on **Debug** **Debugger** to display the Debug Control window. In the Debug Control window, check all four checkboxes (Stack, Source, Locals, and Globals). This makes the Debug Control window provide the most information. Then press **F5** in the file editor window to run the program. This time it will be run under the debugger.

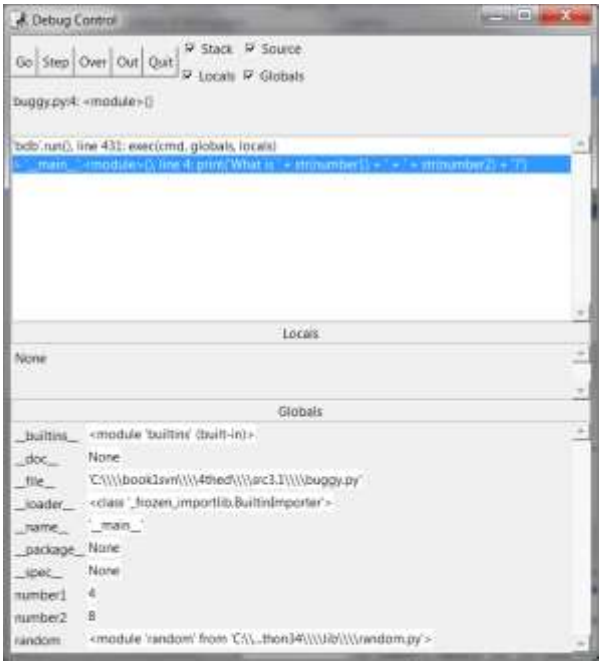
The debugger starts at the line. Nothing special happens here, so just click **Step** to execute it. You will see the module added to the Globals area.

Click **Step** again to run line 2. A new file editor window will appear with the *random.py* file. You have stepped inside the function inside the -in functions be the source of your bugs, so click **Out** to step out of the function and back to your program. Then close the *random.py* file's window.

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

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

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



```
buggy.py:4: <module>()

'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__'.<module>(), line 4: print('What is ' + str(number1) + ' + ' + str(number2) + '?')
```

```
None
```

```
__builtins__ <module 'builtins' (built-in)>
__doc__      None
__file__     'C:\\\\book1svn\\\\4thed\\\\src3.1\\\\buggy.py'
__loader__   <class '_frozen_importlib.BuiltinImporter'>
__name__     '__main__'
__package__  None
__spec__     None
number1      4
number2      8
random       <module 'random' from 'C:\\...thon34\\\\lib\\\\random.py'>
```

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

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

Line 6 is an statement. The condition is that the value in must match the sum of and . If the condition is , then the debugger will move to line 7. If the condition is , the debugger will move to line 9. Click one more time to find out where it goes.

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

That means that would have evaluated to . A string value and an integer value will always not equal each other, so the condition evaluated to .

That is the bug in the program. The bug is that the code has when it should have . Change line 6 to , and run the program again.

```
5
```

This time, the program worked correctly. Run it one more time and enter a wrong answer on "

the computer wil "

you intend.

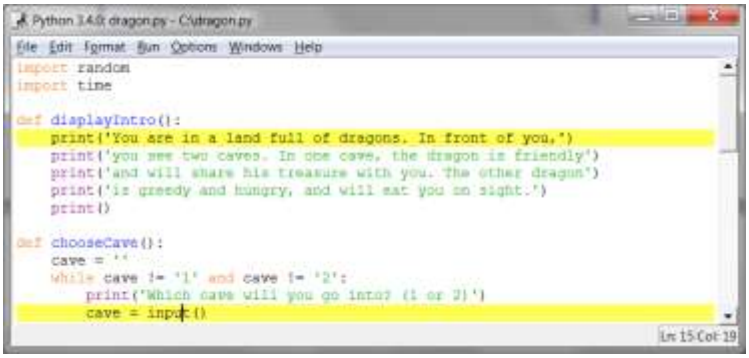
Break Points

program to run at normal speed until it reaches a certain line. A **break point** is set on a line when you want the debugger to take control once execution reaches that line. If you think a problem with your code on, say, line 17, just set a break point on line 17 (or maybe a few lines before that).

When execu "

through lines one at a time to see what is happening. Clicking **Go** will execute the program normally until it reaches another break point or the end of the program.

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


Python 3.4.0: dragon.py - C:\dragon.py
File Edit Format Run Options Windows Help
Ln: 15 Col: 19

```
import random
import time

def displayIntro():
    print('You are in a land full of dragons. In front of you,')
    print('you see two caves. In one cave, the dragon is friendly')
    print('and will share his treasure with you. The other dragon')
    print('is greedy and hungry, and will eat you on sight.')
    print()

def chooseCave():
    cave = ''
    while cave != '1' and cave != '2':
        print('Which cave will you go into? (1 or 2)')
        cave = input()
```

Figure 7-5: The file editor with two break points set.

Example Using Break Points

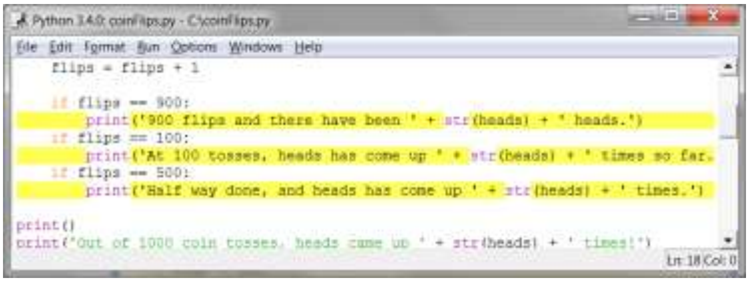
Here is a program that simulates coin flips by calling . The function returning the integer
variable will track how many coin flips have been done. The variable will track how many came up heads.

e thousand times. This would take a person over an hour to do, but the computer can do it in one second! Type in the following code into the file editor and save it as *coinFlips.py*. If you get errors after typing this code in, compare the code you typed to http://invpy.com/diff/coinflips.

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

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

You can click **Step** " through the entire program. Instead, set a break point on lines 12, 14, and 16. The file editor will highlight these lines as shown in Figure 7-6.


Python 3.4.0: coinFlips.py - C:\coinFlips.py
File Edit Format Run Options Windows Help
Ln: 18 Col: 0

```
    flips = flips + 1

    if flips == 900:
        print('900 flips and there have been ' + str(heads) + ' heads.')
    if flips == 100:
        print('At 100 tosses, heads has come up ' + str(heads) + ' times so far.
    if flips == 500:
        print('Half way done, and heads has come up ' + str(heads) + ' times.')

print()
print('Out of 1000 coin tosses, heads came up ' + str(heads) + ' times!')
```

Figure 7-6: Three break points set.

After setting the breakpoints, click **Go** in the Debug Control window. The program will run at normal speed until it reaches the next break point. When is set to , the condition for the statement on line 13 is . This causes line 14 (where 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 and 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 and have changed.

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

Summary

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

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

Topics Covered In This Chapter:

- How to play Hangman
- ASCII art
- Designing a program with flow charts

"

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

How to Play Hangman

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

If they guess correctly, the first player writes the letter in the proper blank. If they guess incorrectly, the first player draws a single body part of the hanging man. If the second player can guess all the letters in the word before the hangman is completely drawn, they win. But if they

" .

Sample Run of Hangman

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

```
a
o
r
t
a
```

c

no

ASCII Art

The graphics for Hangman are keyboard characters printed on the screen. This type of graphics is called **ASCII art** - , which was a sort of precursor to emojii. Here is a cat drawn in ASCII art:

Designing a Program with a Flowchart

This game is a bit more complicated than the ones you seen so far, so take a moment to think
put together
Realm chapter) to help visualize what this program will do. This chapter will go over what flow charts are and why they are useful. The next chapter will go over the source code to the Hangman game.

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

the arrows to other

Figure 8-1 is a complete flow chart for Hangman. You can only move from one box to another in the direction of the arrow. You can never go backwards unless a second arrow going back, like i

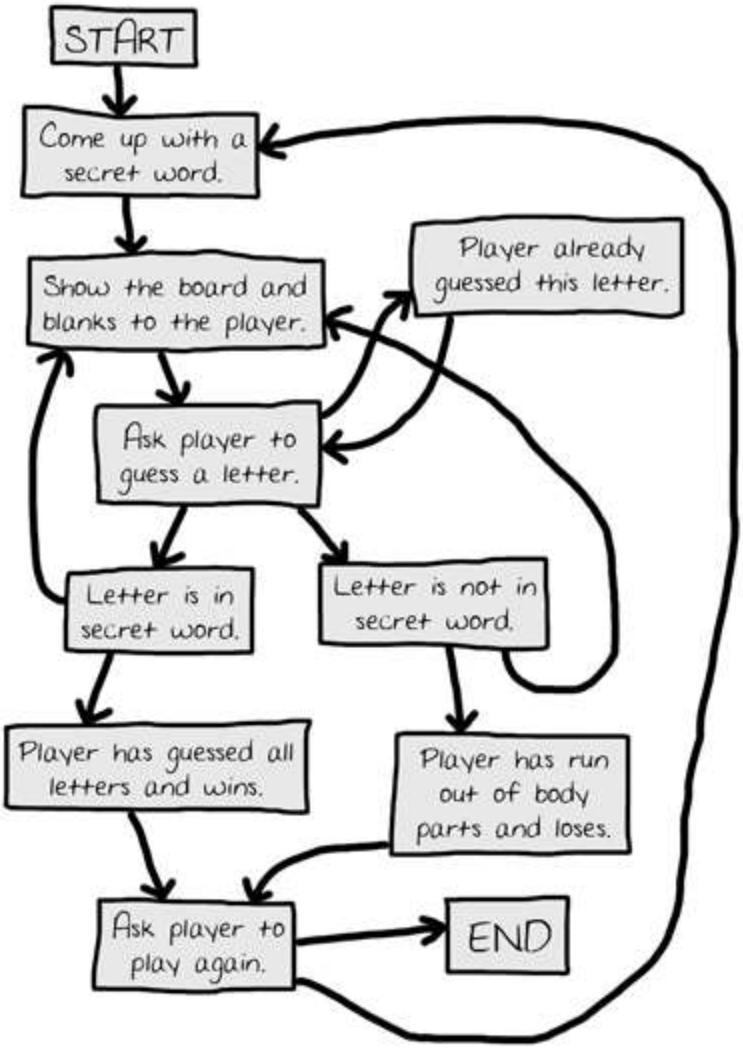

START
Come up with a secret word.
Show the board and blanks to the player.
Player already guessed this letter.
Ask player to guess a letter.
Letter is in secret word.
Letter is not in secret word.
Player has guessed all letters and wins.
Player has run out of body parts and loses.
Ask player to play again.
END

Figure 8-1: The complete flow chart for what happens in the Hangman game.

" *have* to make a flow chart. You could just start writing code. But often once of things that must be added or changed. You may end up

" "
best to plan how the program will work before you start writing it.

Creating the Flow Chart

Your flow **you** understand the flow chart you made, it will be helpful when you start coding. A flow chart that begins with just a " -2:

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

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

The arrows show the order that the program should move. That is, first the program should come up with a secret word, and after that it should ask the player to guess a letter.

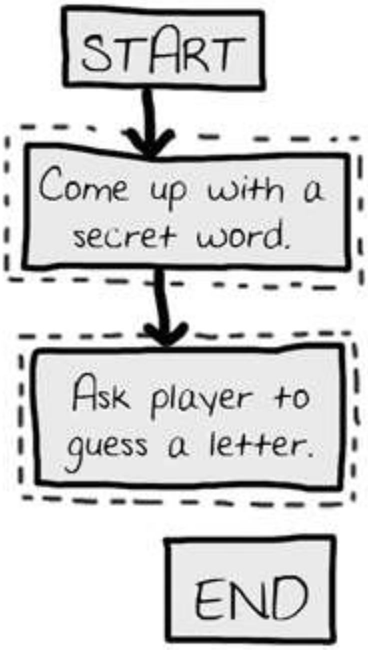


Figure 8-3: Draw out the first two steps of Hangman as boxes with descriptions.

the secret word or not.

Branching from a Flowchart Box

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

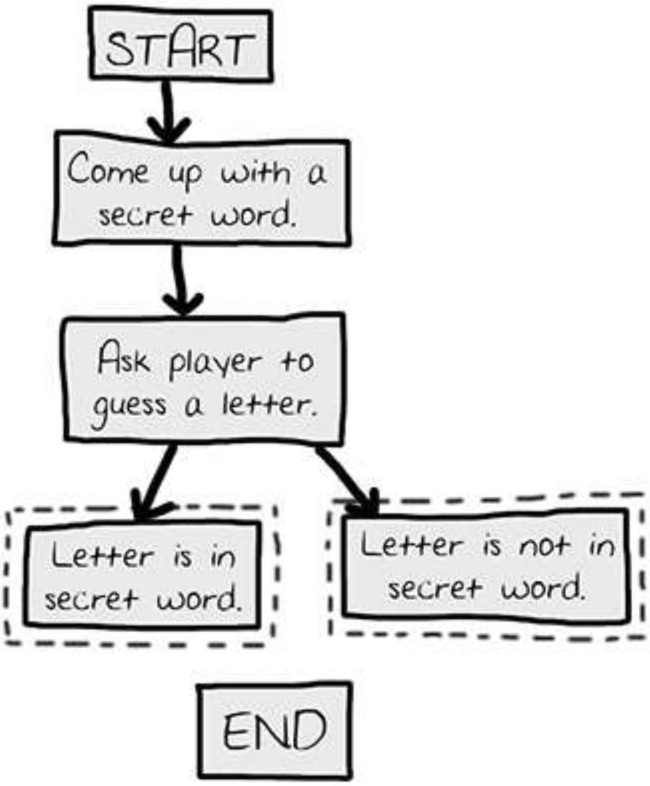

START
Come up with a secret word.
Ask player to guess a letter.
Letter is in secret word.
Letter is not in secret word.
END

Figure 8-4: The branch has two arrows going to separate boxes.

If the letter is in the secret word, check if the player has guessed all the letters and won the game.

"

those cases too.

You **don't**

"

" you

that arrow either. The flow chart now looks like Figure 8-5.

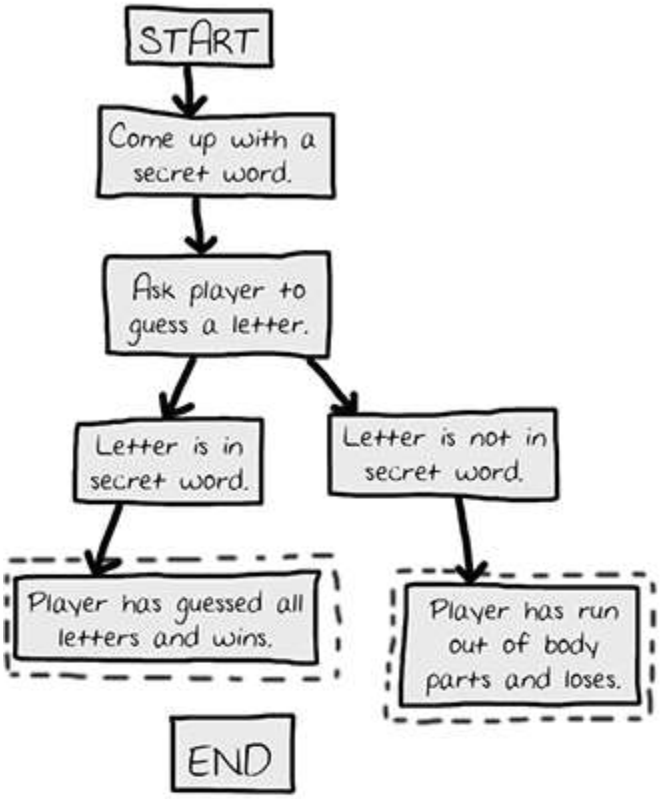

START
Come up with a secret word.
Ask player to guess a letter.
Letter is in secret word.
Letter is not in secret word.
Player has guessed all letters and wins.
Player has run out of body parts and loses.
END

Figure 8-5: After the branch, the steps continue on their separate paths.

Ending or Restarting the Game

Once the player has won or lost, ask them if they want to play again with a new secret word. If " " it thinks up a new secret word. This is shown in Figure 8-6.

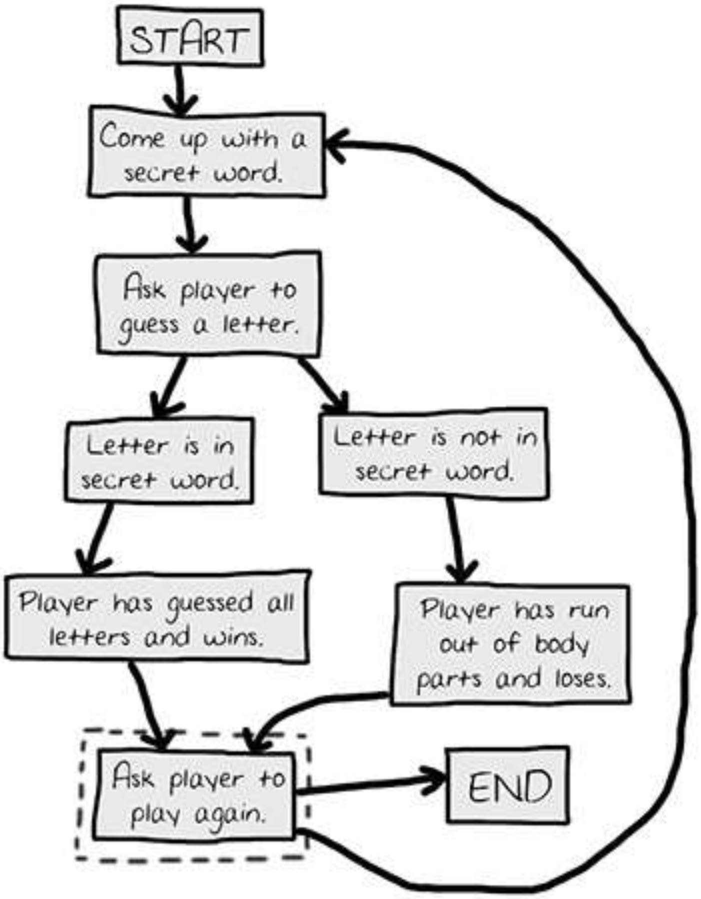

START
Come up with a secret word.
Ask player to guess a letter.
Letter is in secret word.
Letter is not in secret word.
Player has guessed all letters and wins.
Player has run out of body parts and loses.
Ask player to play again.
END

Figure 8-6: The flow chart branches when asking the player to play again.

Guessing Again

" -7.

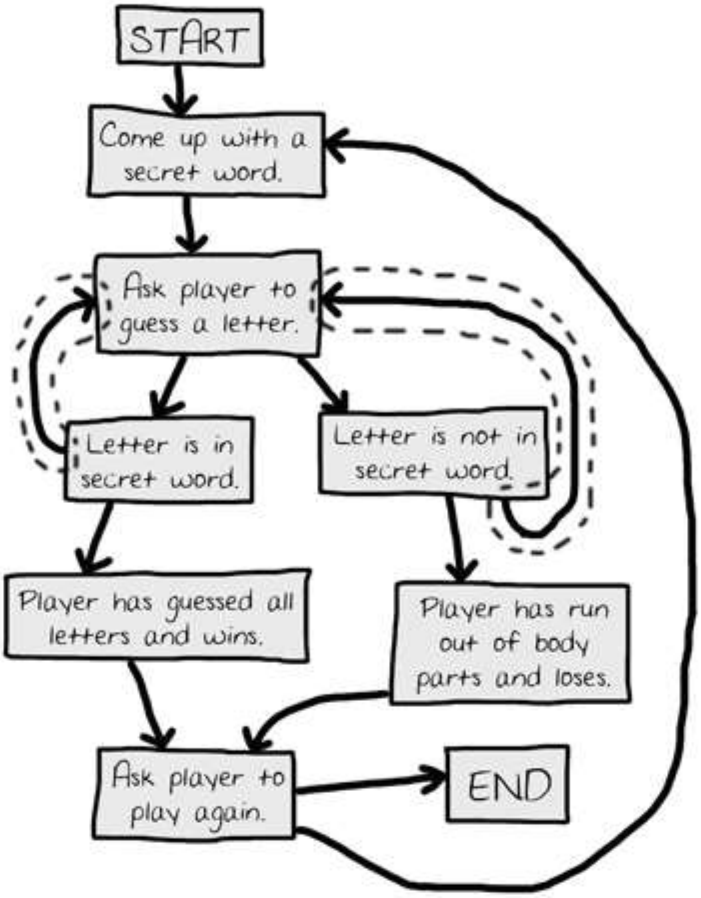

START
Come up with a secret word.
Ask player to guess a letter.
Letter is in secret word.
Letter is not in secret word.
Player has guessed all letters and wins.
Player has run out of body parts and loses.
Ask player to play again.
END

Figure 8-7: The new arrows (outlined) show the player can guess again.

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

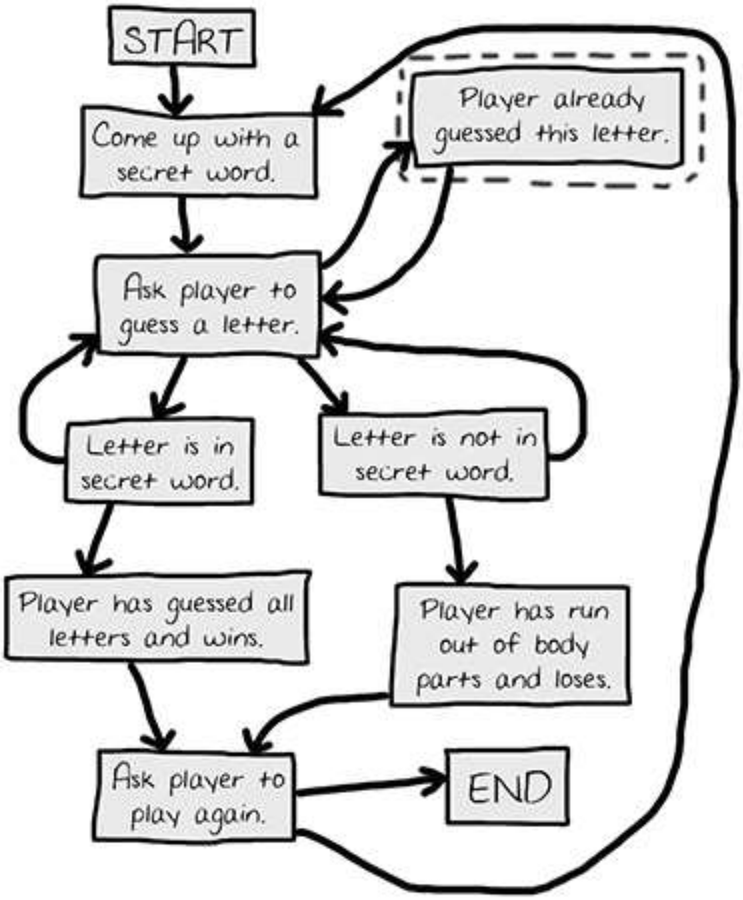

START
Come up with a secret word.
Player already guessed this letter.
Ask player to guess a letter.
Letter is in secret word.
Letter is not in secret word.
Player has guessed all letters and wins.
Player has run out of body parts and loses.
Ask player to play again.
END

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

Offering Feedback to the Player

hangman board and the secret word (with blanks for the letters they haven't guessed yet). These visuals will let them see how close they are to winning or losing the game.

This information is

-9.

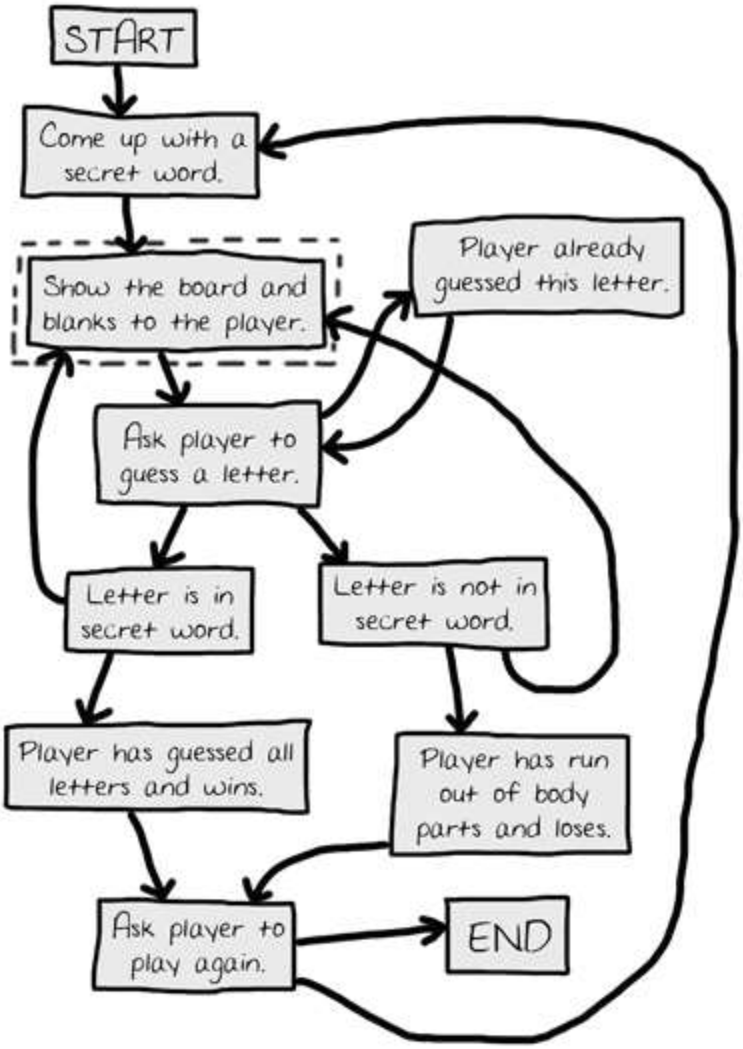

START
Come up with a secret word.
Show the board and blanks to the player.
Player already guessed this letter.
Ask player to guess a letter.
Letter is in secret word.
Letter is not in secret word.
Player has guessed all letters and wins.
Player has run out of body parts and loses.
Ask player to play again.
END

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

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

Summary

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

If you jump in to write the code first, you may discover problems that require you to change the " create new bugs by changing too little or too much. It is much better to know what you want to build before you build it.

Topics Covered In This Chapter:

- Multi-line Strings
- Methods
- Lists
- The and list methods
- The , , , , and string methods
- The and operators
- The and functions
- statements
- loops
- statements

"

"

oop called a loop and a new data type called a list. Once you understand these concepts, it will be much easier to program Hangman.

Source Code of Hangman

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

How the Code Works

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

This one assignment statement stretches over lines 2 to 58 in the source code. To help you
" -line strings.

Multi-line Strings

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

These are **multi-line strings**. In a multi-line string, the newline characters are included as part of
escape character"
use three of them together. This makes the code easier to read for large amounts of text.

Constant Variables

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

" *have* to follow it. But following this convention makes it easier for other programmers to read your code. will always have the value it was assigned on line 2.

Lists

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

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

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

Indexes

Try entering into the interactive shell to store a list in the variable . The square brackets are also used to access an item inside a list. Try entering , , , and into the interactive shell to see how they evaluate:

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

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

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

The evaluation looks like this:

```
          ▼

          ▼

          ▼
```

IndexError

" **IndexError** that will crash your program. Try entering the following into the interactive shell:

Changing the Values of List Items with Index Assignment

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

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

List Concatenation

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

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

The in Operator

The operator can tell you if a value is in a list or not. Expressions that use the operator return a Boolean value: if the value is in the list and
following into the interactive shell:

The expression returns because the string is one of the values in the list. It is located at index .

But if you type the expression , this will return because the string

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

Deleting Items from Lists with del *Statements*

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

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

You can type again and again to keep deleting items from the list:

The "
evaluate to a return value.

Lists of Lists

Lists can contain other values, including other lists "
chores, and a list of your favorite pies. You can put all three lists into another list. Try entering the following into the interactive shell:

To get an item inside the list of lists, you would use two sets of square brackets like this:
which would evaluate to the string .

This is because evaluates to
. That finally evaluates to :

```
▼


▼

▼
```

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

Methods

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

pass a string argument to (as in). You must attach the method call to a specific string value using a period. The next section describes string methods further.

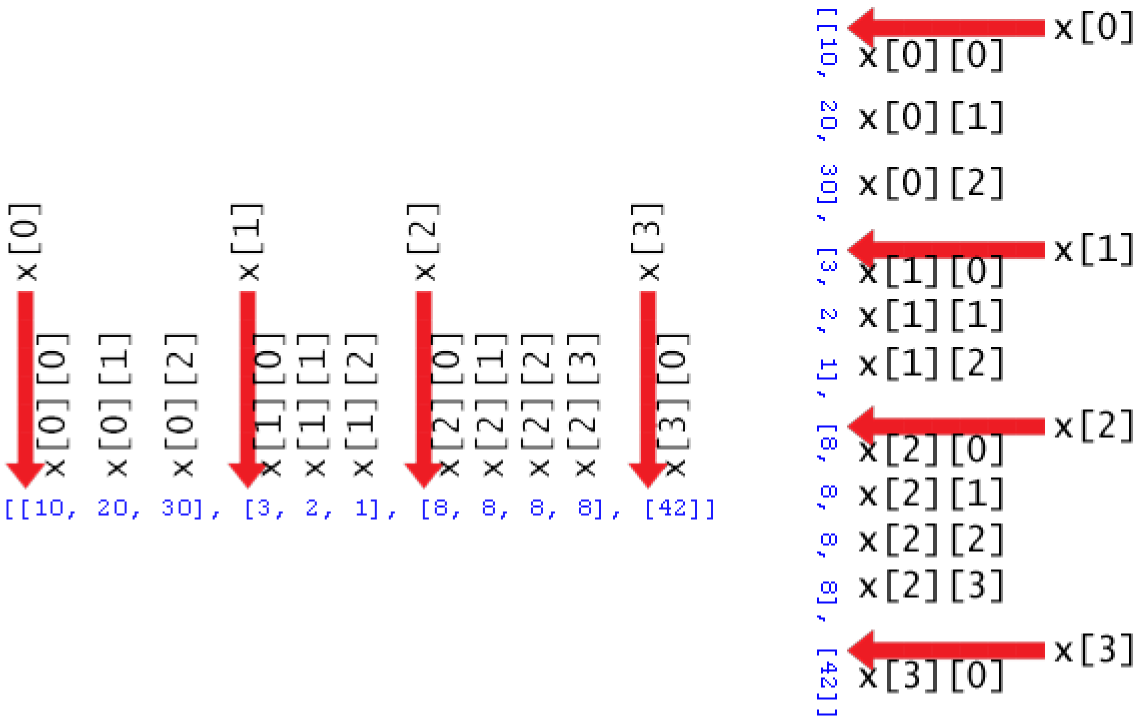


Figure 9-1: The indexes of a list of lists.

The lower() and upper() String Methods

Try entering into the interactive shell to see an example of this method:

There is also an method for strings, which returns a string with all the characters in uppercase. Try entering into the interactive shell:

Because the method returns a string, you can call a method on that string also. Try entering into the interactive shell:

evaluates to the string , and then string's method is called. This returns the string , which is the final value in the evaluation.

```
▼

▼
```

The order is important.
:

That evaluation looks like this:

```
▼

▼
```

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

This does not change the value in . The variable will still contain .

The reverse() and append() List Methods

The list data type also has methods. The method will reverse the order of the items in the list. Try entering , and then to reverse the list. Then enter to view the contents of the variable.

. This method will add the value you pass as an argument to the end of the list. Try entering the following into the interactive shell:

ist. We say that these methods change the list **in-place**.

The split() List Method

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

This assignment statement has just one long string, full of words separated by spaces. And at the end of the string is a method call. The method evaluates to a list with each

It is easier to type the code using . If you created it as a list to begin with, you would have to type: ... and so on, with quotes and commas for every word.

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

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

You can also add your own words to the string on "
the game. Just make sure that spaces separate the words.

How the Code Works

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

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

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

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

was passed as the argument to and that returned the integer . That would mean that line 64 would evaluate to , and then evaluate to . This is how the returns a random string in the list.

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

Displaying the Board to the Player

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

This code defines a new function named . This function has four parameters:

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

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

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

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

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

The range() and list() Functions

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

The function is similar to the or functions. It takes the value it is passed and returns function. Try entering in into the interactive shell:

```
      ...skipped for brevity...
```

The list is so huge, that it even all fit onto the screen. But you can store the list into a variable:

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

The function is often used in loops, which are much like the loops already seen.

for Loops

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

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

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

The range object returned by is equivalent to the list in a statement. The first time the execution goes through the code in the -block, the variable will be set to . On the next iteration, will be set to , and so on.

The statement automatically converts the range object returned by into a list, so no need for in the statement. Just use .

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

A while Loop Equivalent of a for Loop

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

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

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

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

For example, if was this loop would display .

Slicing

List slicing "
indexes (the beginning and end) with a colon in the square brackets after a list. For example, try entering the following into the interactive shell:

The expression evaluates to a list with items from index up to (but not including) index in .

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

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

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

The next part of the code in Hangman uses slicing.

Displaying the Secret Word with Blanks

Now you want code to print the secret word, but with blank lines for the letters that have not been guessed. You can use the _ character (called the underscore character) for this. First create a string with nothing but one underscore for each letter in the secret word. Then replace the blanks for each letter in .

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

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

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

For example, pretend the value of is and the value in is . You would want the string this string.

call would return . The call becomes , which makes the loop iterate over , , , , and .

Because the value of will take on each value in , the code in the loop is the same as this:

If you are confused as to what the value of something like or is, then look at Figure 9-2. It shows the value of the and variables, and the index for each letter in the string.

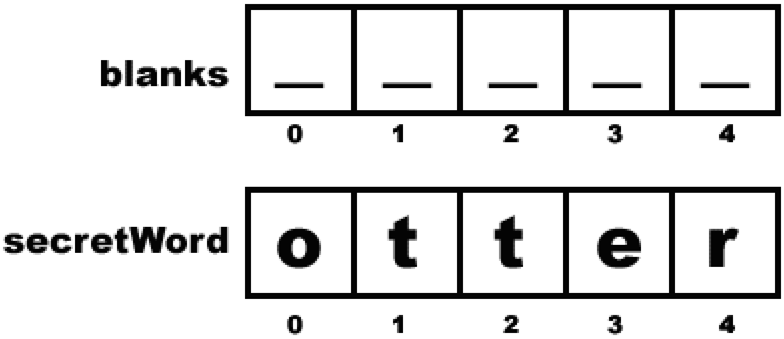


Figure 9-2: The indexes of the and strings.

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

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

Get the Player's Guess

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

A string of the letters the player has guessed is passed as the argument for the parameter. Then the function asks the player to guess a single letter. This single letter will be .

loop will keep asking the player for a letter until they enter text that is:

1. A single letter.
2. A letter they have not guessed previously.

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

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

`elif` ("Else If") Statements

The next part of the Hangman program uses statements. You can think of

" "

none of them are true, do

Take a look at the following code:

If the variable is equal to the string , then the and the if-block tells the user that their cat is fuzzy. However, if this condition is , then Python tries the is , then the string is printed to the screen. If both are , then the code tells or spotted.

You can have as many statements as you want:

When one of the conditions is , its code is executed and then execution jumps to the first line past the else-block. So *one and only one* of the blocks in the - - statements will be executed. You can also leave off the -block if you do one, and just have - statements.

Making Sure the Player Entered a Valid Guess

The variable cont
a valid guess: one and only one lowercase letter. If they didn't, the execution should loop back and ask them for a letter again.

is not one charac
already exists inside the is
not a lowercase letter.

If all of these conditions are , then the
returns the value in on line 98.

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

Asking the Player to Play Again

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

```
▼

▼

▼
```

The point of the function is to let the player type in yes or no to tell the program if they want to play another round of Hangman. The player should be able to type YES, yes, Y, or
If the player types in YES, then the return value of is the string . And returns the lowercase version of the attached string. So the return value of is .

" . This function returns if the associated string begins with the string parameter between the parentheses, and if it
is .

Now you have evaluated this expression! What it does is let the player type in a response, lowercases the response, checks if it begins with the letter , then returns if it does and

On a side note, also an string method that will return if the string ends with the string in and is sort of like the opposite of .

Review of the Hangman Functions

his game!

- will take a list of strings passed to it, and return one string from it. That is how a word is chosen for the player to guess.
- will show the current state of the board, including how much of the secret word the player has guessed so far and the wrong letters the player has guessed. This function needs four parameters passed to work correctly. is a list of strings that hold the ASCII art for each possible hangman board. and are strings made up of the letters that the player has guessed that are in and not in the secret word, respectively. And is the secret word the player is trying to guess. This function has no return value.
- takes a string of letters the player has already guessed and .) This function returns the string of the valid letter the player guessed.
- is a function that asks if the player wants to play another round of Hangman. This function returns if the player does and

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

Setting Up the Variables

Line 106 is the first call that executes when the game is run. It displays the title of the game. Next is assigning blank strings for and since the player rs yet.

The call will evaluate to a randomly selects word from the list.

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

Displaying the Board to the Player

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

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

Letting the Player Enter Their Guess

The function needs all the letters in and combined, so line 116 concatenates the strings in these variables and passes the result as the argument. This argument is needed by because the function has to check if the player types in a letter that they have already guessed.

Checking if the Letter is in the Secret Word

If the string exists in , then concatenate to the end of the string. This string will be the new value of .

Checking if the Player has Won

How can the program know if the player has guessed every letter in the secret word? Well, has each letter that the player correctly guessed and is the secret word itself. But you because consider this case: if was the string and was the string , then would be even though the player *has* guessed each letter in the secret word.

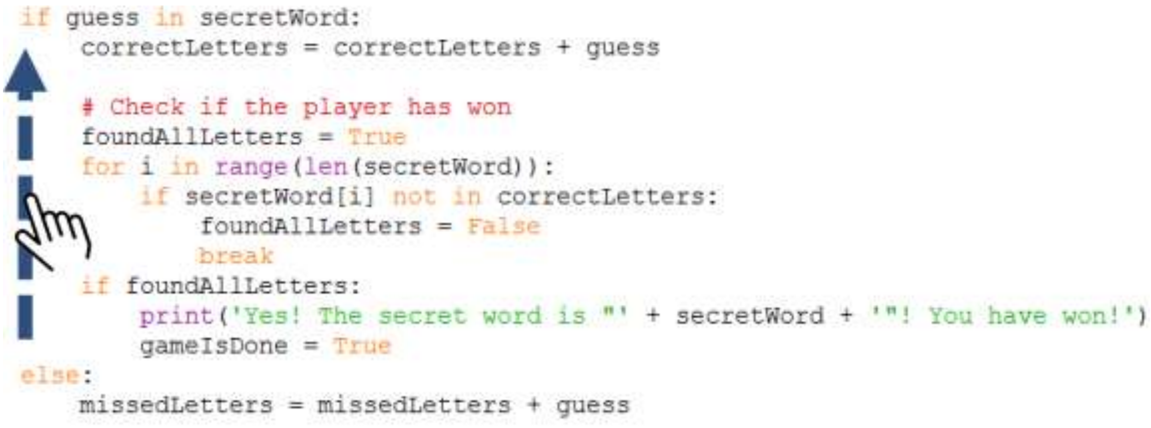
The only way you can be sure the player won is to iterate over each letter in and see if it exists in . If, and only if, every letter in exists in will the player have won.

If you find a letter in , you know that the player has **not** guessed all the letters. The new variable is set to on line 122 before the loop begins. The loop starts out assuming that all the letters in the secret word have on line 125 will change to the first time it finds a letter in .

If all letters in the secret word have been found, the player is told they have won and is set to .

When the Player Guesses Incorrectly

This is the start of the else-block. Remember, the code in this block will execute if the condition was . But which condition? To find out, point your finger at the start of the keyword and move it straight up like in Figure 9- keyword's indentation is the same as the keyword's indentation on line 118.



```
if guess in secretWord:
    correctLetters = correctLetters + guess

    # Check if the player has won
    foundAllLetters = True
    for i in range(len(secretWord)):
        if secretWord[i] not in correctLetters:
            foundAllLetters = False
            break
    if foundAllLetters:
        print('Yes! The secret word is "' + secretWord + '"! You have won!')
        gameIsDone = True
else:
    missedLetters = missedLetters + guess
```

Figure 9-3: The statement is matched with the statement at the same indentation.

So if the condition on line 118 ( ) was , then the execution moves into this else-block.

Wrongly guessed letters are concatenated to the string on line 131. This is like what line 119 did for letters the player guessed correctly.

Each time the player guesses wrong, the code concatenates the wrong letter to the string in . So the length of (or, in code,) is also the number of wrong guesses.

The list has 7 ASCII art strings. So when equals , you know the player has lost because the hangman picture will be finished. Remember, is the first item in the list, and is the last one.

So, when the length of the string is equal to (that is,), the player has run out of guesses. Line 136 prints the secret word and line 137 sets the variable is set to .

If the player won or lost after guessing their letter, the game should ask the player if they want to play again. The function handles getting a yes or no from the player, so it is called on line 141.

If the player does want to play again, the values in and must be reset to blank strings, to , and a new secret word stored in . This way when the execution loops back to the beginning of the loop on line 112, the board will be back to a fresh game.

If the player did not type asked if they wanted to play " and the else-block executes. The statement causes the execution to jump to the first instruction after the loop. But because there are no more instructions after the loop, the program terminates.

Summary

"

"

idea to sketch out a flow chart on paper of what happens in your program.

Lists are values that can contain other values. Methods are functions specific to a data type. Lists have and methods. Strings have , , , , and methods about many more data types and methods in the rest of this book.

The loop is a loop that iterates over the items in a list, unlike a loop which iterates as long as a condition is . The of your - statements. The statement can delete variables or items inside lists.

Topics Covered In This Chapter:

- The dictionary data type
- Key-value pairs
- The and dictionary methods
- Multiple variable assignment

The Hangman is much bigger than the Dragon Realm program, but is also more sophisticated. It really helps to make a flow chart or small sketch to remember how you want everything to work. " an extend it with new features.

" □ many of the words. You can easily give the player more guesses by adding more multi-line strings to the list.

Save your *hangman.py* program as *hangman2.py*, then add the following instructions:

There are two new multi-line strings to the list, one with the hangman's left ear drawn, and the other with both ears drawn. Because the program will tell the player they have lost on line134 based on , this is the only change you must make. The rest of the program works with the new list just fine.

You can also change the list of words by changing the words on line 59. Instead of animals, you could have colors:

Or shapes:

Or fruits:

Dictionaries

With some modification, you can change the code so that the Hangman game uses sets of words, such as animal, color, shape, or fruit. The program can tell the player which set (animal, color, shape, or fruit) the secret word is from.

To make this change, you will need a new data type called a **dictionary**. A dictionary is a collection of values like a list is. But instead of accessing the items in the dictionary with an integer index, you can access them with an index of any data type. For dictionaries, these indexes are called **keys**.

Dictionaries use { and } curly braces instead of [and] square brackets. Try entering the following into the interactive shell:

The values between the curly braces are **key-value pairs**. The keys are on the left of the colon and

y using the key.

Try entering the following into the interactive shell:

Instead of putting an integer between the square brackets, you can use, say, a string key. This will evaluate to the value for that key.

Getting the Size of Dictionaries with len()

You can get the number of key-value pairs in the dictionary with the function. Try entering the following into the interactive shell:

The Difference Between Dictionaries and Lists

Dictionaries can have keys of any data type, not just strings. But remember, because and are different values, they will be different keys. Try entering this into the interactive shell:

The keys in dictionaries can also be looped over using a loop. Try entering the following into the interactive shell.

Dictionaries are different from lists because the values inside them are unordered. The first item in a list named would be . But " because dictionaries do not have any sort of order. Try entering the following into the interactive shell:

The expression evaluates to because dictionaries are unordered and considered equal if they have the same key-value pairs in them. Meanwhile, lists are ordered, so two lists with the same values in a different order are not equal to each other. Try entering this into the interactive shell:

Dictionaries have two useful methods, and . These will return values of a type called and , respectively. Much like range objects, values of those data types are returned by the function. Try entering the following into the interactive shell:

Sets of Words for Hangman

"

replace the value assigned to with a dictionary whose keys are strings and values are lists of strings. The string method will return a list of strings with one word each.

Lines 59 to 62 are across multiple lines in the source code, but they are still one assignment statement. The instruction until the final } curly brace on line 62.

The random.choice() Function

The function in the module takes a list argument and returns a random value from it. This is similar to the what the previous function did in the new version of the function.

To see how the function works, try entering the following into the interactive shell:

Change the function so that its parameter will be a dictionary of lists of strings, instead of just a list of strings. Here is what the function originally looked like:

Change the code in this function so that it looks like this:

The name of the parameter is changed to to be more descriptive. Now instead of choosing a random word from a list of strings, first the function chooses a random key in the dictionary by calling .

And instead of returning the string , the function returns a list with two items. The first item is . The second item is .

Evaluating a Dictionary of Lists

The expression on line 72 may look complicated, but it is just an expression you can evaluate one step at a time like anything else. First, imagine that had the value (which was chosen on line 65) and has the value 5 (chosen on line 68). Here is how would evaluate:

```
▼

▼


▼


▼
```

In the above case, the item in the list this function returns would be the string . (Remember that indexes start at , so refers to the 6 th item in the list, not the 5th.)

Because the function now returns a list of two items instead of a string, will be assigned a list, not a string. You can assign these two items into two separate variables using multiple assignment. This is explained next.

Multiple Assignment

Multiple assignment is a shortcut to specify multiple variables, separated by commas, on the left side of an assignment statement. Try entering the following into the interactive shell:

The above example is equivalent to the following assignment statements:

You must put the same number of variables on the left side of the assignment operator as there are items in the list on the right side. Python will automatically assign the first item's value in the list to the first variable, the second item's value to the second variable, and so on. But if you do not have the same number of variables and items, the Python interpreter will give you an error.

Change your code in Hangman on line 109 and 145 to use multiple assignment with the return value of :

Printing the Word Category for the Player

The last change

way, when the player plays the game they will know if the secret word is an animal, color, shape, or fruit. Add this line of code after line 112. Here is the original code:

Add the line so your program looks like this:

the changes to the Hangman program. Instead of just a single list of strings, the secret word is chosen from many different lists of strings. The program also tells the player which set of words the secret word is from. Try playing this new version. You can easily change the dictionary on line 59 to include more sets of words.

Summary

" n always add more features after you learn more about Python programming.

Dictionaries are similar to lists except that they can use any type of value for an index, not just integers. The indexes in dictionaries are called keys.

Multiple assignment is a shortcut to assign multiple variables the values in a list.

Hangman was fairly advanced compared to the previous games in this book. But at this point, you know most " " " data types such as lists and dictionaries. The later programs in this book will still be a challenge to master, but you have finished the steepest part of the climb.

Topics Covered In This Chapter:

- Artificial Intelligence
- List References
- Short-Circuit Evaluation
- The Value

This chapter features a Tic Tac Toe game against a simple artificial intelligence. An **artificial intelligence** (or **AI**) is

Tic Tac Toe is really just a few lines of code.

Two people play Tic Tac Toe with paper and pencil. One player is X and the other player is O. Players take turns placing their X or O. If a player gets three of their marks on the board in a row, column or one of the two diagonals, they win. When the board fills up with neither player winning, the game ends in a draw.

This chapter introduce many new programming concepts. It makes use of our existing

at a sample run of the program. The player makes their move by entering the number of the space they want to go. These numbers are in the same places as the number keys on your keyboard's keypad (see Figure 10-2).

Sample Run of Tic Tac Toe

```
X
```

```
3
4
5
no
```

Source Code of Tic Tac Toe

In a new file editor window, type in the following source code and save it as *tictactoe.py*. Then run the game by pressing **F5**.

Designing the Program

Figure 10-1 is what a flow chart of Tic Tac Toe could look like. In the Tic Tac Toe computer program the player chooses if they want to be X or O. Who takes the first turn is randomly chosen. Then the player and computer take turns making moves.

side shows what happens on the computer's turn. After the player or computer makes a move, the program checks if they won or caused a tie, and then the game switches turns. After the game is over, the program asks the player if they want to play again.

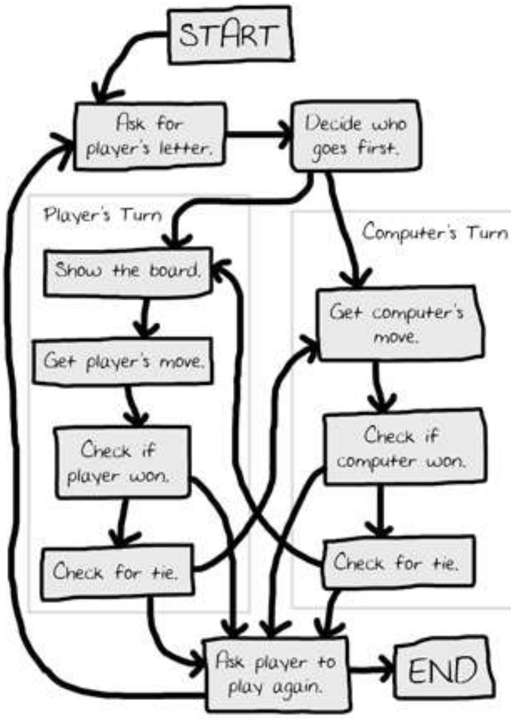


Figure 10-1: Flow chart for Tic Tac Toe

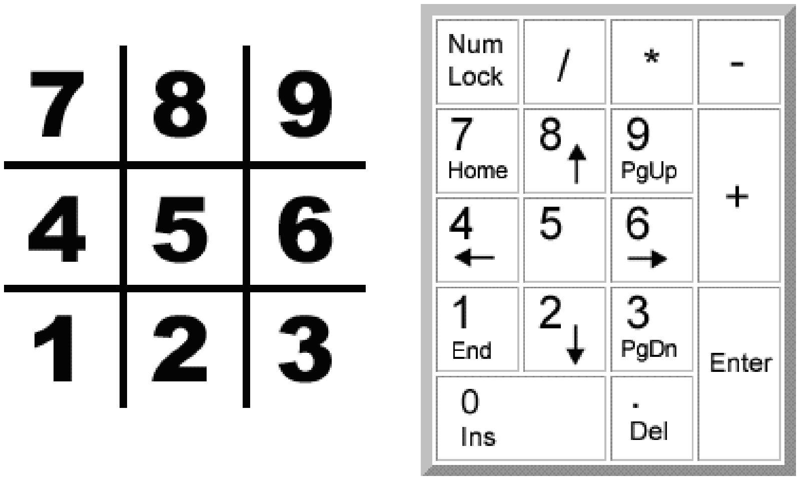


Figure 10-2: The board is numbered like the keyboard's number pad.

Representing the Board as Data

First, you must figure out how to represent the board as data in a variable. On paper, the Tic Tac Toe board is drawn as a pair of horizontal lines and a pair of vertical lines, with either an X, O, or empty space in each of the nine spaces.

In the program, the Tic Tac Toe board is represented as a list of strings. Each string will represent one of the nine spaces on the board. To make it easier to remember which index in the list is for which space, they will mirror th "
10-2.

The strings will either be for the X player, for the O player, or a single space for a blank space.

So if a list with ten strings was stored in a variable named , then would be the top-left space on the board.] would be the center. would be the left side space, and so on. The program will ignore the string at index in the list. The player will enter a number from 1 to 9 to tell the game which space they want to move on.

Game AI

The AI needs to be able to look at a board and decide which types of spaces it will move on. To be clear, we will label three types of spaces on the Tic Tac Toe board: corners, sides, and the center. Figure 10-3 is a chart of what each space is.

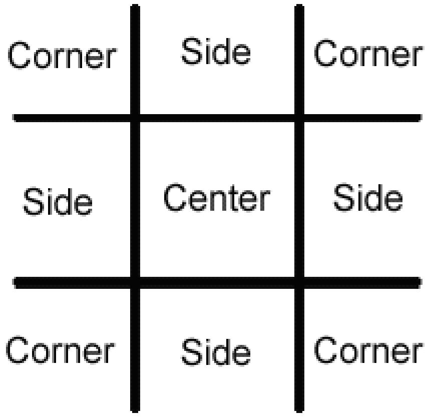


Figure 10-3: Locations of the side, corner, and center places.

algorithm is a finite series of instructions to compute a result. A single program can make use of several different algorithms. An algori
will compute the best move to make, as shown in Figure 10-4.

1. First, see if 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 a move the player can make that will cause the computer to lose the game. If there is, move there to block the player. Otherwise, go to step 3.
3. Check if any of the corner spaces (spaces 1, 3, 7, or 9) are free. If so, move there. If no corner piece is free, then go to step 4.
4. " "
5. Move on any of the side pieces (spaces 2, 4, 6, or 8). There are no more steps, because if the execution reaches step 5 the side spaces are the only spaces left.

□ the flow chart in Figure 10-1. You could add this information to the flow chart with the boxes in Figure 10-4.

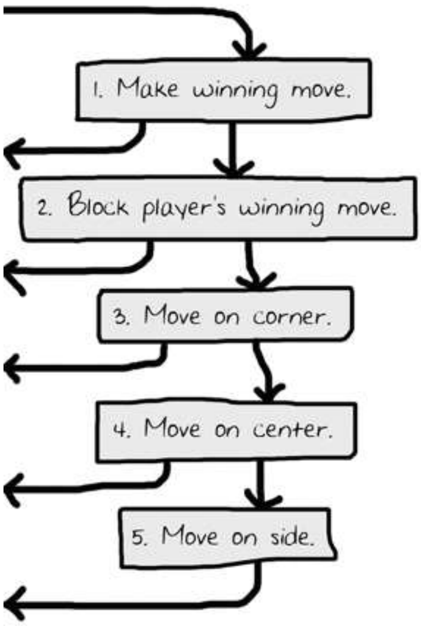


Figure 10-4: The five steps of the "Get computer's move" algorithm. The arrows leaving go to the "Check if computer won" box.

This algorithm is implemented in the function and the other functions that calls.

The Start of the Program

The first couple of lines are a comment and importing the module so you can call the function.

Printing the Board on the Screen

The function will print the game board represented by the parameter. Remember that the board is represented as a list of ten strings, where the string at index is the mark on space 1 on the Tic Tac Toe board, and so on. The string at index is ignored. Many of as the board.

Be sure to get the spacing right in the strings, otherwise the board will look funny when printed on the screen. Here are some example calls (with an argument for) to and what the function would print:

Letting the Player be X or O

The function asks if the player wants to be X or O. It will keep asking the player until the player types in an X or O. Line 27 automatically changes the string returned by the call to to uppercase letters with the string method.

The on contains parentheses, which means the expression inside the parentheses is evaluated first. If the variable was set to , the expression would evaluate like this:

```
▼
▼
▼
▼
▼
```

If has the value or " and lets the program execution continue past the while-block.

This function returns a list with two items. The first item (the string at index letter, and the second item (the string at index) is the computer's letter. These - statements chooses the appropriate list to return.

Deciding Who Goes First

The function does a virtual coin flip to determine whether the computer or the player goes first. The coin flip is in calling . If this function call returns a , the function returns the string . Otherwise, the function returns the string . The code that calls this function will use the return value to know who will make the first move of the game.

Asking the Player to Play Again

The function asks the player if they want to play another game. The function returns if the player types in , , , or anything that begins with the letter Y. For any other response, the function returns . This function is identical to the one in the Hangman game.

Placing a Mark on the Board

The function is simple and only one line. The parameters are a list with ten strings named " or) named , and a place on the board where that player wants to go (which is an integer from to) named .

But wait a second. This code seems to change one of the items in the list to the value in . But because this code is in a function, the parameter will be forgotten when the function re be forgotten as well?

"

about the difference between lists and references to lists.

References

Try entering the following into the interactive shell:

These results make sense from what you know so far. You assign to the variable, and then assign the value in and to the variable When you later overwrite to , . This is because and are different variables that store different values.

sign, you are actually assigning a list reference to the variable. A **reference** is a value that points to some bit of data. Here is some code that will make this easier to understand. Type this into the interactive shell:

This looks odd. The code only changed the list, but it seems that both the and lists have changed. This is because the variable does not contain the list value itself, but rather contains a reference to the list as shown in Figure 10-5. The actual list itself is not contained in any variable, but rather exists outside of them.

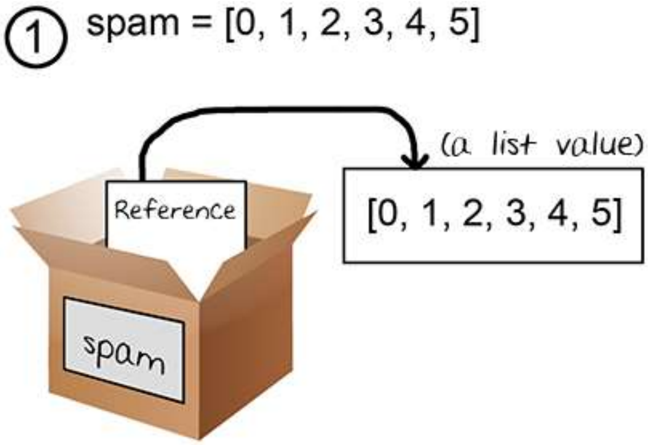


Figure 10-5: Variables don’t store lists, but rather references to lists.

Notice that copies the *list reference* in to , instead of copying the list value itself. Now both and store a reference that refers to the same list value. But there is only one list. The list was not copied, the reference to the list was copied. Figure 10-6 shows this copying.

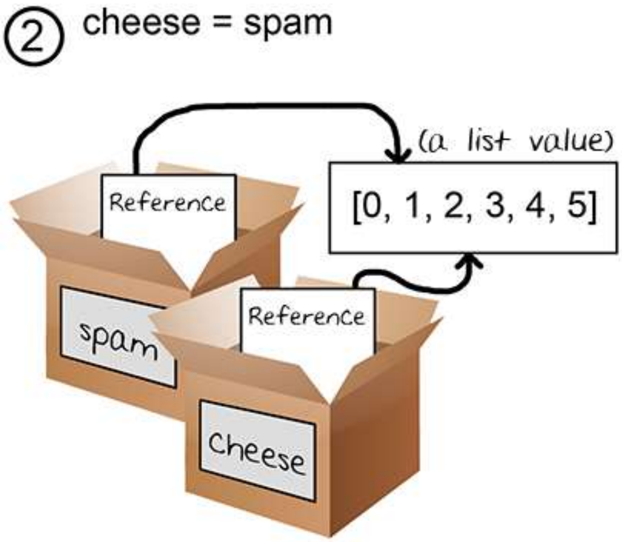


Figure 10-6: Two variables store two references to the same list.

So the line changes the same list that refers to. This is why seems to have the same list value that does. They both have references that refer to the same list, as shown in Figure 10-7.

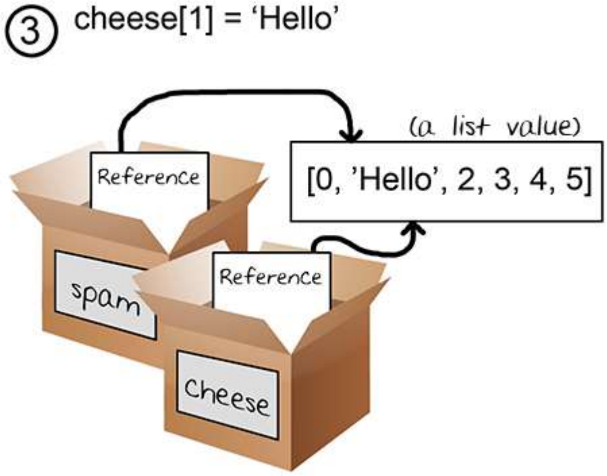


Figure 10-7: Changing the list changes all variables with references to that list.

If you want and to store two different lists, you have to create two different lists instead of copying a reference:

In the above example, and have two different lists stored in them (even though these lists are identical in content). Now if you modify one of the lists, it affect the other because and have references to two different lists:

Figure 10-8 shows how the two references point to two different lists.

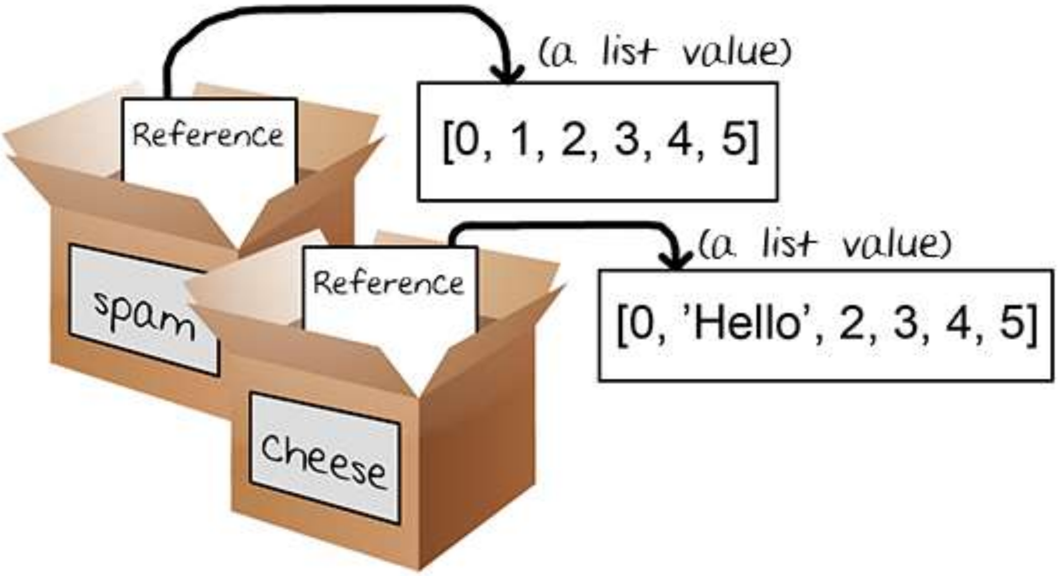


Figure 10-8: Two variables each storing references to two different lists.

"

dictionaries.

Using List References in makeMove()

function:

When a list value is passed for the parameter, the function's local variable is really a copy of the reference to the list, not a copy of the list. But a copy of the reference still refers to the same list the original reference refers. So any changes to in this function will also happen to the original list. Even though is a local variable, the function modifies the original list.

The and parameters are copies of the string and integer values that you pass. Since they are copies of values, if you modify or in this function, the original variables you used when you called

Checking if the Player Has Won

Lines 53 to 60 in the function are actually one long statement. The and names are shortcuts for the and parameters. These shorter names mean you have "

There are eight possible ways to win at Tic Tac Toe. You can have a line across the top, middle, and bottom rows. Or you can have a line down the left, middle, or right columns. Or you can have a line over either of the two diagonals.

Note that each line of the condition checks if the three spaces are equal to the letter provided (combined with the operator) and you use the operator to combine the eight different ways to win. This means only one of the eight ways must be true in order for us to say that the player who owns letter in is the winner.

is and is

. The board looks like this:

Here is how the expression after the keyword on line 53 would evaluate:

First Python will replace the variables and with the value inside of them:

Next, Python will evaluate all those comparisons inside the parentheses to a Boolean value:

Then the Python interpreter will evaluate all those expressions inside the parentheses:

Since now only one value inside the parentheses, you can get rid of them:

Now evaluate the expression that is connecter by all those operators:

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

So given those values for and , the expression would evaluate to . This is how the program can tell if one of the players has won the game.

Duplicating the Board Data

The function is here so that you can easily make a copy of a given 10-string list that represents a Tic Tac Toe board in the game. There are times that ll want the AI algorithm to make temporary modifications to a temporary copy of the board without changing the original board. In that case, call this function to make a copy of the board's list. The new list is created on line 64, with the blank list brackets .

But the list stored in on line 64 is just an empty list. The loop will iterate over the parameter, appending a copy of the string values in the original board to the duplicate board. Finally, after the loop, is returned. The function builds up a copy of the original board and returning a reference to this new board in , and not the original one in .

Checking if a Space on the Board is Free

This is a simple function that, given a Tic Tac Toe board and a possible move, will return if that move is available or not. Remember that free spaces on the board lists are marked as a single
"

Letting the Player Enter Their Move

The function asks the player to enter the number for the space they want to move on. The loop makes sure the execution does not continue until the player has entered an integer from 1 to 9. It also checks that the space entered "
Toe board passed to the function for the parameter.

The two lines of code inside the loop simply ask the player to enter a number from 1 to 9. The condition on line 78 is if either of the expressions on the *left* or *right* side of the operator is .

The expression on the *left* side checks if the move is equal to , , , and so on up to by creating a list with these strings (with the method) and checking if is in this list.

evaluates to
, but the former easier to type.

The expression on the *right* side checks if the move that the player entered is a free space on the board. It checks this by calling the function. Remember that will return if the move you pass is available on the board. Note that expects an integer for , so the function returns an integer form of .

The operators are added to both sides so that the condition is when either of these requirements are unfulfilled. This will cause the loop to ask the player again and again until they enter a proper move.

Finally, line 81 returns the integer form of whatever move the player entered. Remember that returns strings, so the function is called to return an integer form of the string.

Short-Circuit Evaluation

You may have noticed a possible problem in the function. What if the player typed in or some other non-integer string? The expression
on the left side of would return as expected, and then Python would evaluate the expression on the right side of the operator.

But calling would cause an error. Python gives this error because the function can only take strings of number characters, like or , not strings like .

As an example of this kind of error, try entering this into the interactive shell:

But when you play the Tic Tac Toe game and try entering " happen. The reason is because the s being short-circuited.

Short-circuiting means is that since the part on the left side of the keyword ( ) evaluates to , the Python interpreter knows that the entire expression will evaluate to . matter if the expression on the right side of the keyword evaluates to or , because only one value on the side of the operator needs to be .

Think about it: The expression evaluates to and the expression also evaluates to . If the value on the left side is , the value is on the right side:

always evaluates to

always evaluates to

part. This means the and the functions are never called as long as is .

This works out well for the program, because if the right side is then number form. That would cause to give us an error. The only times evaluates to are when single-digit string. In that case, the call to would not give us an error.

An Example of Short-Circuit Evaluation

a short program that gives a good example of short-circuiting. Try entering the following into the interactive shell:

When is called, it prints and then also displays the return value of . The same goes for .

Now try entering the following into the interactive shell.

The first part makes sense: The expression calls both of the functions, so you see both of the printed messages.

But the second expression only shows but not . This is because Python did not call at all. Since the left side of the operator is " returns and The evaluation was short-circuited.

The same applies for the operator. Try entering the following into the interactive shell:

If the left side of the operator is , then the entire expression is whether the right side of the operator is or , so Pyth it. Both and evaluate to , so Python short-circuits the evaluation.

Choosing a Move from a List of Moves

The function is useful for the AI code later in the program. The parameter is a list of strings that represents a Tic Tac Toe board. The second parameter is a list of integers of possible spaces from which to choose. For example, if is , that means should return the integer for one of the corner spaces.

However, will first check that the space is valid to make a move on. The list starts as a blank list. The loop will iterate over . The moves that cause to return are added to with the method.

At this point, the list has all of the moves that were in that are also " at least one possible move that can be made on the board.

But this list could be empty. For example, if was but the board represented by the parameter had all the corner spaces already taken, the list would be . In that case, will evaluate to and the function returns the value . This next section explains the value.

The None Value

The **None value** is a value that represents the lack of a value. is the only value of the data type . It can be useful to use the value when you need a value that mea

For example, say you had a variable named True-False pop quiz question. The variable could hold or You could set to would be better because otherwise it may look like the user answered the question when they didn't.

Functions that return by reaching the end of the function (and not from a statement) have for a return value. The

As a side note, will not be displayed in the interactive shell like other values will be:

Functions that seem to return anything actually return the value. For example, returns :

Creating the Computer's Artificial Intelligence

The function . The first argument is a Tic Tac Toe board for the parameter. The second argument is letter for the computer either or in the parameter. The first few lines simply assign the other letter to a variable named . This way the same code can be used whether the computer is X or O.

The function will returns an integer from to representing the computer move.

Remember how the Tic Tac Toe AI algorithm works:

- First, see if a move the computer can make that will win the game. If there is, take that move. Otherwise, go to the second step.

- Second, see if a move the player can make that will cause the computer to lose the game. If there is, the computer should move there to block the player. Otherwise, go to the third step.
- Third, check if any of the corner spaces (spaces 1, 3, 7, or 9) are free. If no corner space is free, then go to the fourth step.
- Fourth, check if the center is free. If so" "
- Fifth, move on any of the side pieces (spaces 2, 4, 6, or 8). There are no more steps, because if the execution has reached this step then the side spaces are the only spaces left.

The Computer Checks if it Can Win in One Move

More than anything, if the computer can win in the next move, the computer should make that winning move immediately. The loop that starts on line 105 iterates over every possible move from 1 to 9. The code inside the loop will simulate what would happen if the computer made that move.

The first line in the loop (line 106) makes a copy of the list. This is so the simulated move variable. The returns an identical but separate board list value.

Line 107 checks if the space is free and if so, simulates making the move on the copy of the board. If this move results in the computer winning, the function returns

If none of the spaces results in winning, the loop will finally end and the program execution continues to line 113.

The Computer Checks if the Player Can Win in One Move

Next, the code will simulate the human player moving on each of the spaces. The code is similar

function shows that the player would win with this move, then the computer will return that same move to block this from happening.

If the human player cannot win in one more move, the loop will eventually finish and execution continues to line 121.

Checking the Corner, Center, and Side Spaces (in that Order)

The call to with the list of will ensure that it returns the integer for one of the corner spaces: , , , or . If all the corner spaces are taken, the function will return and execution moves on to line 126.

If none of the corners are available, line 127 moves on the center space if it is free. If the center
" e execution moves on to line 130.

This code also makes a call to , except you pass it a list of the side spaces ( ). This function because the side spaces are the only spaces that can possibly be left. This ends the function and the AI algorithm.

Checking if the Board is Full

The last function is . This function returns if the 10-string list board argument it was passed has an or in every index (except for index , which is ignored). If at least one space in that is set to a single space then it will return .

The loop will let us check indexes 1 through 9 on the list. As soon as it finds a free space on the board (that is, when returns) the function will return .

If execution manages to go through every iteration of the loop, then none of the spaces are free. Line 137 will then execute .

The Start of the Game

"

when you run this program. It greets the player.

loop has for the condition and will keep looping until the execution encounters a statement. Line 144 sets up the main Tic Tac Toe board in a variable named . It is a 10-string list, where each string is a single space .

Rather than type out this full list, line 144 uses list replication. It is shorter to type then .

Deciding the Player's Mark and Who Goes First

The function lets the player type in whether they want to be X or O. The function returns a 2-string list, either or . The multiple assignment trick will set to the first item in the returned list and to the second.

The function randomly decides who goes first, and returns either the string or the string and line 147 tells the player who will go first. The variable keeps track of whether the game is still being played or if someone has won or tied.

Running the Player's Turn

computer's turn, as long as is set to .

The variable was originally set by the call on line 146. It is set either to or . If equals " condition is and execution jumps to line 169.

Line 153 calls and passes the variable to print the Tic Tac Toe board on the screen. Then the function lets the player type in their move (and also makes sure it is a valid move). The .

Now that the player has made their move, the computer should check if they have won the game with this move. If the function returns , the if-
board and prints a message telling them they have won.

The variable is also set to
computer's turn.

their last move, maybe their move filled up the entire board and tied the game. In this else-block, the function returns if there are no more moves to make. In that case, the if-block starting at line 162 displays the tied board and tell the player a tie has occurred. The execution breaks out of the loop and jumps to line 186.

" line 167 sets the variable to so that it will ex s turn on the next iteration.

Running the Computer's Turn

If the for the condition on line 151, then it must be the computer's turn. The code in this else-block is similar to

Lines 170 to 184 are almost identical to

.

"line 184 sets to

code inside the loop, so execution would jump back to the statement on line 150.

Lines 186 and 187 are located immediately after the while-block started by the statement on line 150. is set to when the game has ended, so at this point the game asks the player if they want to play again.

If returns , then the (because the operator reverses the Boolean value) and the statement executes. That breaks the execution out of the loop that was started on line 142. But since there are no more lines of code after that while-block, the program terminates.

Summary

Creating a program that can play a game comes down to carefully considering all the possible situations the AI can be in and how it should respond in each of those situations. The Tic Tac Toe AI is simple because there are not many possible moves in Tic Tac Toe compared to a game like chess or checkers.

Our AI checks if any possible move can allow itself to win. Otherwise, it checks if it must block Then the AI simply chooses any available corner space, then the center space, then the side spaces. This is a simple algorithm for the computer to follow.

The key to implementing our AI is by making copies of the board data and simulating moves on the copy. That way, the AI code can see if a move results in a win or loss. Then the AI can make that move on the real board. This type of simulation is effective at predicting what is a good move or not.

Topics Covered In This Chapter:

- Augmented Assignment Operators, , , ,
- The Function
- The and List Methods
- String Interpolation (also called String Formatting)
- Conversion Specifier
- Nested Loops

"

learn about augmented assignment operators and string interpolation. These things do anything you couldn't do before, but they are nice shortcuts to make coding easier.

Bagels is a deduction game you can play with a friend. Your friend thinks up a random 3-digit number with no repeating digits, and you try to guess what the number is. After each guess, your friend gives you three types of clues:

- **Bagels** None of the three digits you guessed is in the secret number.
- **Pico** One of the digits is in the secret number, but your guess has the digit in the wrong place.
- **Fermi** Your guess has a correct digit in the correct place.

You can get multiple clues after each guess. If the secret number is 456 and your guess is 546 the

Sample Run of Bagels

```
123
```

```

453

425

326

489

075

015

175

no
```

Source Code of Bagels

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

Designing the Program

The flow chart in Figure 11-1 describes what happens in this game, and in what order they can happen.

How the Code Works

At the start of the program, import the module. Then define a function named . The function makes a secret number that has only unique digits in it. Instead of only 3-digit secret numbers, the parameter lets the function make a secret number with any number of digits. For example, you can make a secret number of four or six digits by passing or for .

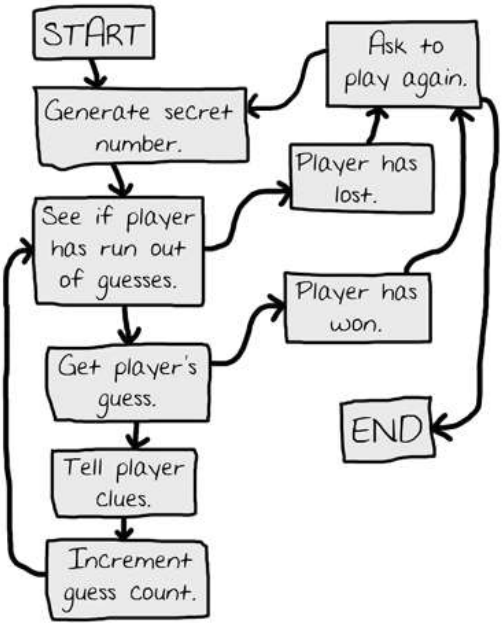


Figure 11-1: Flow chart for the Bagels game.

Shuffling a Unique Set of Digits

always evaluate to
easier to type . The variable contains a list of all ten digits.

The random.shuffle() Function

The function randomly changes the order of . This function
" modifies
the function in the Tic Tac Toe chapter modified the list it was passed in place, rather than return a new list with the change. This is why you do **not** write code like
.

Try experimenting with the function by entering the following code into the interactive shell:

You want the secret number in Bagels to have unique digits. The Bagels game is much more fun digits in the secret number, such as or . The function will help you do this.

Getting the Secret Number from the Shuffled Digits

The secret number will be a string of the first digits o-3( )11(t)6(he s)-3(h)11(uf)7( )11(-4(ne)9(d l)6(

Augmented Assignment Operators

The operator on line 8 is one of the **augmented assignment operators**. Normally, if you wanted to add or concatenate a value to a variable, you would use code that looked like this:

The augmented assignment operators are a shortcut that frees you from retyping the variable name. The following code does the same thing as the above code:

"the code must figure out what clues to give the player. The list in will start empty and have and strings added as needed.

Do this by looping through each possible index in and . The strings in both variables will be the same length, so the line 18 could have used either or and work the same. As the value of changes from to to , and so on, line 19 checks if the first, second, third, etc. letter of is the same as the number in the same index of . If so, line 20 will add a string to clue.

Otherwise, line 21 will check if the number at the th position in guess exists anywhere in . If so, you know that the number is somewhere in the secret number but not in the same position. Line 22 will then add to .

If the list is empty after the loop, then you know that there are no correct digits at all in . In this case, line 24 returns the string as the only clue.

The sort() List Method

Lists have a method named that rearranges the items in the list to be in alphabetical or numerical order. Try entering the following into the interactive shell:

The "
This is just like how the method works.

You would never want to use this line of code: because that would return the value (which is what returns). Instead you would want a separate line and then the line .

The reason you want to sort the list is to get rid of extra information based on the order of the clues. If was , then that would tell the player that the center digit of the guess is in the correct position. Since the other two clues are both Pico, the player would know that all they have to do is swap the first and third digit to get the secret number.

" sure which number the Fermi clue refers. This is what we want for the game.

The join() String Method

The helps determine if the player entered a valid guess. Line 31 checks if is the blank string, and if so, returns .

The loop iterates over each character in the string . The value of will have a single character on each iteration. Inside the for-block, the code checks if returned by . (The return value from is equivalent to but is easier to type.) " you know a non-digit character in . In that case, line 36 returns .

If execution continues past the loop, then you know that every character in is a digit. In that case, line 38 returns .

Finding out if the Player Wants to Play Again

After all of the function definitions, this is the actual start of the program. Instead of using the integer in our program for the number of answer has, use the constant variable . The same goes for using the constant variable instead of the integer for the number of guesses the player gets. Now it will be easy to change the number of guesses or secret number digits. Just change line 45 or 46 and the rest of the program will still work without any more changes.

The function calls will tell the player the rules of the game and what the Pico, Fermi, and Bagels clues mean. Line 48's call has added to the end and inside the string. This is a technique known as string interpolation.

String Interpolation

String interpolation is a coding shortcut. Normally, if you want to use the string values inside variables in another string, you have to use the concatenation operator:

As you can see, it can be hard to type a line that concatenates several strings. Instead, you can use **string interpolation**, which lets you put placeholders like . These placeholders are called **conversion specifiers**. Then put all the variable names at the end. Each is replaced with a variable at the end of the line. For example, the following code does the same thing as the previous code:

String interpolation can make your code much easier to type. The first variable name is used for the first , the second variable with the second and so on. You must have the same number of conversion specifiers as you have variables.

Another benefit of using string interpolation instead of string concatenation is interpolation works with any data type, not just strings. All values are automatically converted to the string data type. If you concatenated an integer to a string" d get this error:

String concatenation can only combine two strings, but is an integer. You would have to remember to put instead of . But with string interpolation, this conversion to strings is done for you. Try entering this into the interactive shell:

String interpolation is also known as **string formatting**.

Creating the Secret Number

Line 55 is an infinite loop that has a condition of so it will loop forever until a statement is executed. Inside the infinite loop, you get a secret number from the function, passing it to tell how many digits you want the secret number to have. This secret number is assigned to . Remember, the value in is a string not an integer.

Line 57 tells the player how many digits is in the secret number by using string interpolation instead of string concatenation. Line 59 sets variable to to mark this is as the first guess. Then line 60 has a new loop that loops as long as is less than or equal to .

Getting the Player's Guess

The . The code keeps looping and asking the player for a guess until the player enters a valid guess. A valid guess has only digits and the same number of digits as the secret number. This is what the loop that starts on line 62 is for.

The variable is set to the blank string on line 61 so the the first time it is checked, ensuring the execution enters the loop.

Getting the Clues for the Player's Guess

After execution gets past the loop that started on line 62, contains a valid guess. Pass this and to the function. It returns a string of the clues, which are displayed to the player on line 67. Line 68 increments using the augmented assignment operator for addition.

Checking if the Player Won or Lost

Notice that this second loop on line 60 is inside another loop that started on line 55. These loops-inside-loops are called **nested loops**. Any or statements will only break or continue out of the innermost loop, and not any of the outer loop.

If is the same value as , the player has correctly guessed the secret number and line 71 breaks out of the loop that was started on line 60.

If not, then execution continues to line 72, where it checks if the player ran out of guesses. If so,

At this point, execution jumps back to the loop on line 60 where it lets the player have another guess. If the player ran out of guesses (or it broke out of the loop with the statement on line 71), then execution would proceed past the loop and to line 75.

Asking the Player to Play Again

Line 75 asks the player if they want to play again by calling the function. If returns , break out of the loop that started on line 55. Since no more code after this loop, the program terminates.

If returned , then the execution would not execute the statement and execution would jump back to line 55. The program generates a new secret number so the player can play a new game.

Summary

Bagels is a simple game to program but can be difficult to win at. But if you keep playing, yo eventually discover better ways to guess and make use of the clues the game gives you. This is

This chapter introduced a few new functions and methods (, , and), along with a couple handy shortcuts. An augmented assignment operators involve less typing whe such as in , which can be shortened to . String interpolation can make your code much more readable by placing (called a conversion specifier) inside the string instead of using many string concatenation operations.

"

to create in the later chapters of this book. We will learn about the math concepts of Cartesian coordinates and negative numbers. These are used in the Sonar, Reversi, and Dodger games, but Cartesian coordinates and negative numbers are used in many games. If you already know about these concepts, give the next chapter a brief read anyway to refresh yourself.

Topics Covered In This Chapter:

- Cartesian coordinate systems
- The X-axis and Y-axis
- The Commutative Property of Addition
- Absolute values and the function

. Instead it goes over some simple mathematical concepts you will use in the rest of this book. In 2D games the graphics on the screen can move left or right and up or down. These two directions make up two-dimensional, or 2D, space. Games with objects moving around a two-dimensional computer screen need a way to translate a place on the screen into integers the program can deal with.

This is where the Cartesian coordinate system comes in. The coordinates are numbers for a
ables.

Grids and Cartesian Coordinates

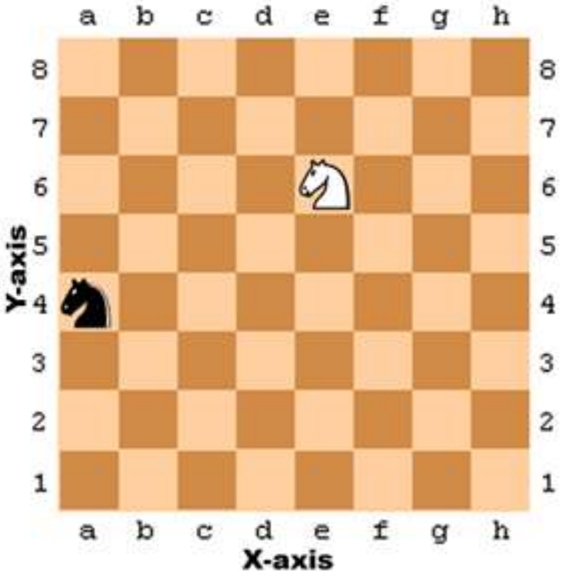


Figure 12-1: A sample chessboard with a black knight at a, 4 and a white knight at e, 6.

A common way to refer to specific places on a chessboard is by marking each row and column with letters and numbers. Figure 12-1 is a chessboard that has each row and each column marked.

A coordinate for a space on the chessboard is a combination of a row and a column. In chess, the knight piece looks like a horse head. The white knight in Figure 12-1 is located at the point e, 6 and the black knight is located at point a, 4.

This labeled chessboard is a Cartesian coordinate system. By using a row label and column label, you

Subtract the black knight - s X-coordinate: 5 - 1 = 4. The black knight has to move along the X-axis by four spaces.

Subtract the black knight s Y-coordinate and white knight s Y-coordinate: 6 - 4 = 2. The black knight has to move along the Y-axis by two spaces.

By doing some math with the coordinate numbers, you can figure out the distances between two coordinates.

Negative Numbers

Cartesian coordinates use negative numbers. **Negative numbers** are numbers that are smaller than zero. A minus sign in front of a number shows it is negative. -1 is smaller than 0. And -2 is smaller than -1. If you think of regular numbers (called **positive numbers**) as starting from 1 and increasing, you can think of negative numbers as starting from -1 and decreas
positive or negative. In Figure 12-3, you can see the positive numbers increasing to the right and the negative numbers decreasing to the left on a number line.

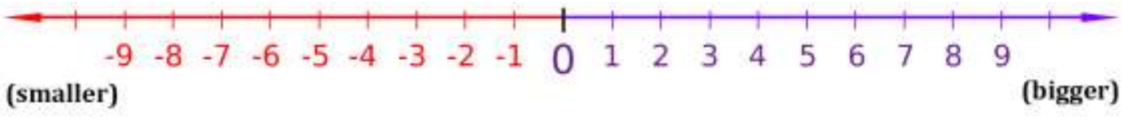


Figure 12-3: A number line.

The number line is useful to see subtraction and addition done with negative numbers. The expression 4 + 3 can be thought of as the white knight starting at position 4 and moving 3 spaces over to the right (addition means increasing, which is in the right direction).

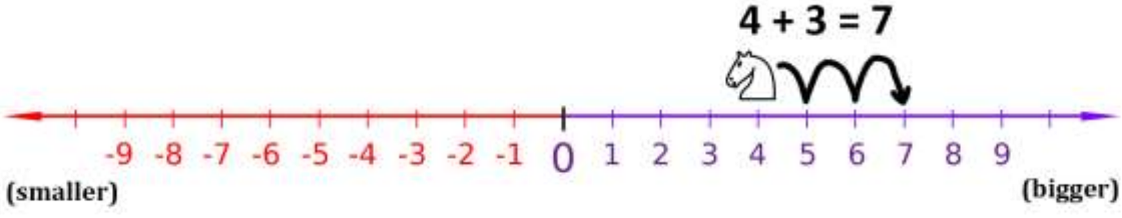


Figure 12-4: Moving the white knight to the right adds to the coordinate.

As you can see in Figure 12-4, the white knight ends up at position 7. This makes sense, because 4 + 3 is 7.

Subtraction is done by moving the white knight to the left. Subtraction means decreasing, which is in the left direction. 4 - 6 would be the white knight starting at position 4 and moving 6 spaces to the left, like in Figure 12-5.

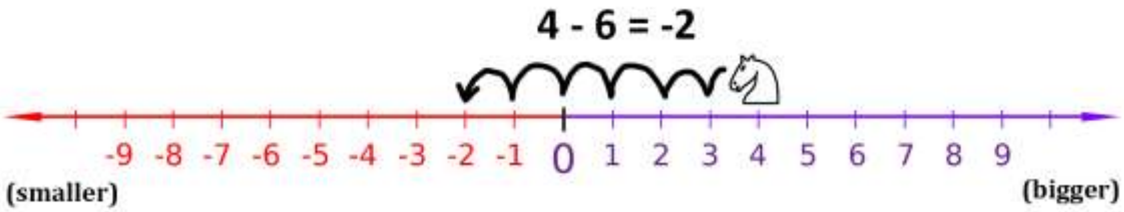


Figure 12-5: Moving the white knight to the left subtracts from the coordinate.

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

If you add or subtract a negative number, the white knight would move in the *opposite* direction. If you add a negative number, the knight moves to the *left*. If you subtract a negative number, the knight moves to the *right*. The expression -6 - -4 would be equal to -2. The knight starts at -6 and moves to the *right* by 4 spaces. Notice that -6 - -4 has the same answer as -6 + 4.

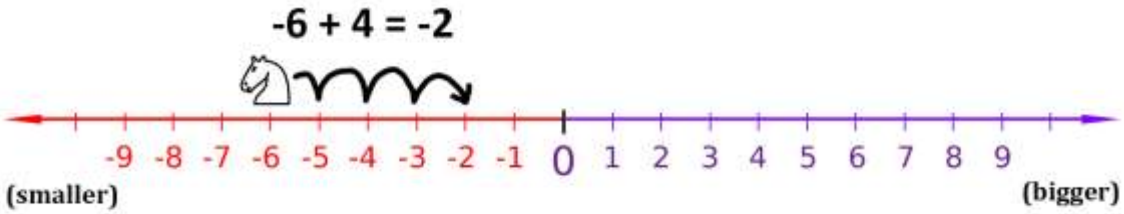


Figure 12-6: Even if the white knight starts at a negative coordinate, moving right still adds to the coordinate.

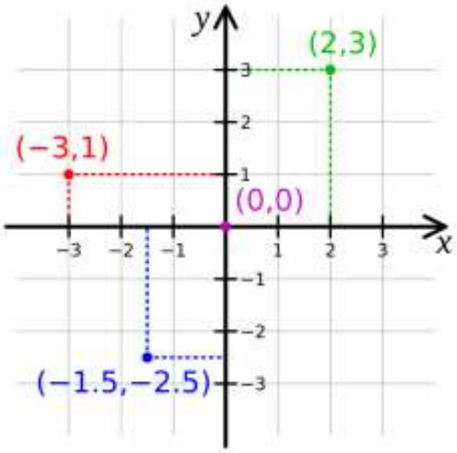


Figure 12-7: Putting two number lines together creates a Cartesian coordinate system.

You can think of the X-axis as a number line. Add another number line going up and down for the Y-axis. If you put these two number lines together, you have a Cartesian coordinate system like in Figure 12-7.

Adding a positive number (or subtracting a negative number) would move the knight up the number line, and subtracting a positive number (or adding a negative number) would move the knight down.

The 0, 0 coordinate is called the **origin**.

Math Tricks

Subtracting and adding negative numbers is easy when you have a number line in front of you. It can also be easy without a number line too. Here are three tricks to help you add and subtract negative numbers by yourself.

Trick 1: "A Minus Eats the Plus Sign on its Left"

When you see a minus sign with a plus sign on the left, you can replace the plus sign with a minus sign. The answer is still the same, because adding a negative value is the same as subtracting a positive value. 4 + -2 and 4 - 2 both evaluate to 2.

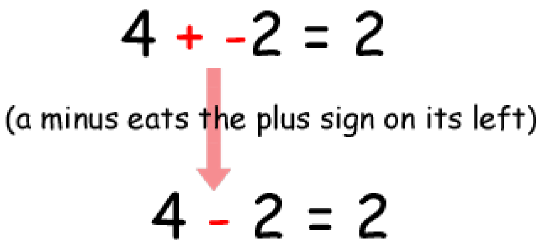


Figure 12-8: Trick 1 - Adding a positive and negative number.

Trick 2: "Two Minuses Combine Into a Plus"

When you see the two minus signs next to each other without a number between them, they can combine into a plus sign. The answer is still the same, because subtracting a negative value is the same as adding a positive value.

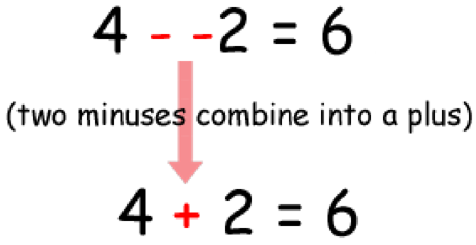


Figure 12-9: Trick 2 - Subtracting a positive and negative number.

Trick 3: The Commutative Property of Addition

You can always swap the numbers in addition. This is the **commutative property** of addition. That means that doing a swap like 6 + 4 to 4 + 6 will not change the answer.

If you count the boxes in Figure 12-10" if you swap the numbers for addition.

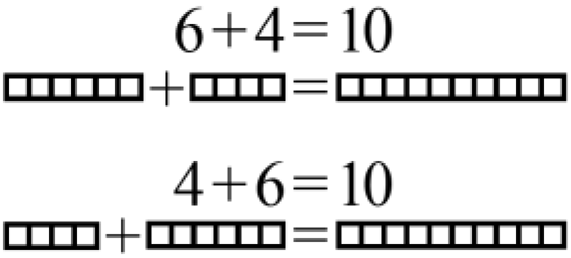


Figure 12-10: Trick 3 - The commutative property of addition.

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. -6 + 8 is the same as 8 + -6.

Then when you look at 8 + -6, you see that the minus sign can eat the plus sign to its left, and the problem becomes 8 - 6 = 2. But this means that -6 + 8 is also 2!
to have the same answer, but made it easier for us to solve without using a calculator or computer.

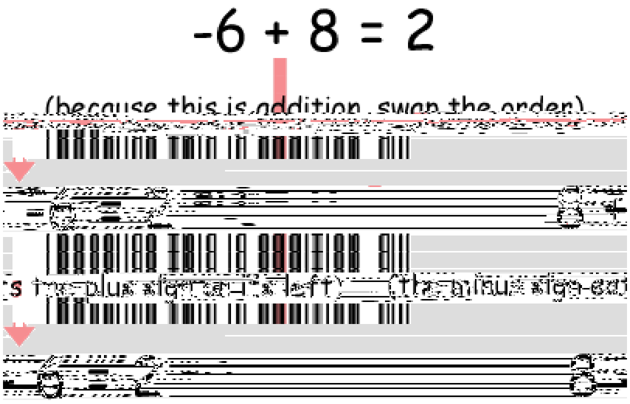


Figure 12-11: Using the math tricks together.

Absolute Values and the abs() Function

The **absolute value** of a number is the number without the negative sign in front of it. Therefore, positive numbers do not change, but negative numbers become positive. For example, the absolute value of -4 is 4. The absolute value of -7 is 7. The absolute value of 5 (which is positive) is just 5.

You can figure out the distance between two objects by subtracting their positions and taking the absolute value of the difference. Imagine that the white knight is at position 4 and the black knight is at position -2. The distance would be 6, since 4 - -2 is 6, and the absolute value of 6 is 6.

It works no matter what the order of the numbers is. -2 - 4 (that is, negative two minus four) is -6, and the absolute value of -6 is also 6.

function returns the absolute value of an integer. Try entering the following into the interactive shell:

Coordinate System of a Computer Screen

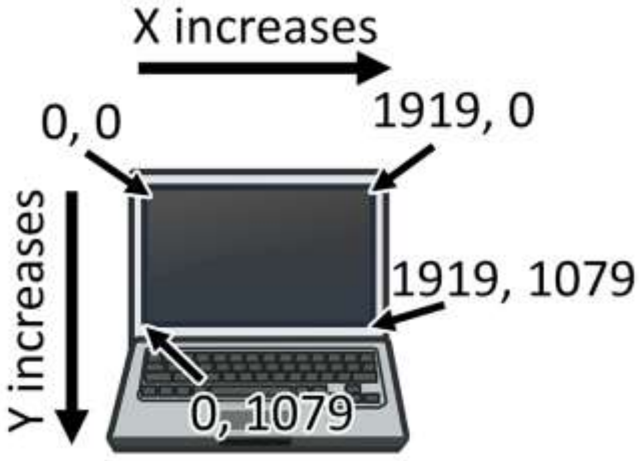


Figure 12-12: The Cartesian coordinate system on a computer screen.

It is common that computer screens use a coordinate system that has the origin (0, 0) at the top left corner of the screen, which increases going down and to the right. This is shown in Figure 12-12. There are no negative coordinates. Most computer graphics use this coordinate system, and you will use it in games.

Summary

M "
getting by on simple addition and multiplication.

Cartesian coordinate systems are needed to describe where in a two-dimensional area a certain position is. Coordinates have two numbers: the X-coordinate and the Y-coordinate. The X-axis runs left and right and the Y-axis runs up and down. On a computer screen, origin is in the top-left corner and the coordinates increase going right and down.

The three tricks you learned in this chapter make it easy to add positive and negative integers. The first trick is that a minus sign will eat the plus sign on its left. The second trick is that two minuses next to each other will combine into a plus sign. The third trick is that you can swap the position of the numbers you are adding.

For the rest of the book, we will use the concepts from this chapter in our games because they have two-dimensional areas in them. All graphical games require understanding how Cartesian coordinates work.

Topics Covered In This Chapter:

- Data structures
- The list method
- The string method
- The function

The game in this chapter is the first to make use of Cartesian coordinates that you learned about in Chapter 12. The game also has **data structures** (which is just a fancy way of saying complex variables such as those that contain lists of lists.) As the games you program become more

"

" ces at various places in the ocean to locate sunken treasure chests. Sonar is a technology that ships use to locate objects under the sea. The sonar devices (in this game) will tell the player how far away the closest treasure chest is, but not in what direction. But by placing multiple sonar devices down, the player can figure out where the treasure chest is.

There are three chests to collect, but the player has only sixteen sonar devices to use to find them. Imagine that you could not see the treasure chest in the following picture. Because each sonar device can only find the distance, not direction, the possible places the treasure could be is anywhere in a square ring around the sonar device (see Figure 13-1).

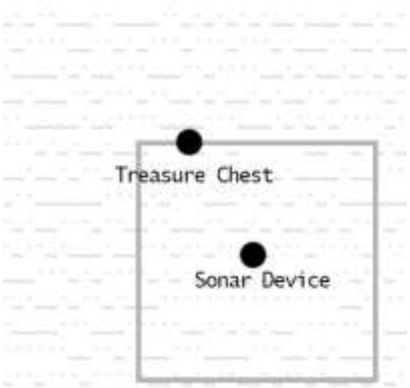


Figure 13-1: The sonar device’s square ring touches the (hidden) treasure chest.

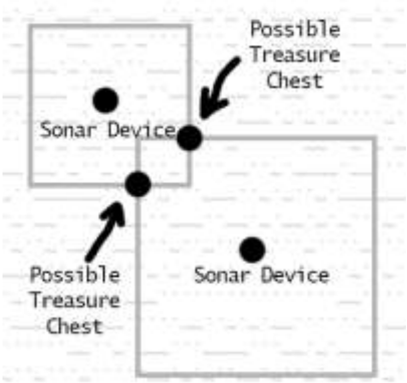


Figure 13-2: Combining multiple square rings of shows where treasure chests could be.

But multiple sonar devices working together can narrow it to an exact place where the rings intersect each other. See Figure 13-2. (Normally these rings would be circles, but this game will use squares to make programming it easier.)

Sample Run of Sonar Treasure Hunt

```
no
10 10
```

```
15 6
```

```
15 10
```

```
no
```

Source Code of Sonar Treasure Hunt

Below is the source code for the game. Type it into a new file, then save the file as *sonar.py* and run it by pressing the **F5** key. If you get errors after typing this code in, compare the code you typed http://invpy.com/diff/sonar.

Designing the Program

Before trying to understand the source code, play the game a few times first to understand what is going on. The Sonar game uses lists of lists and other such complicated variables, called **data structures**. Data structures are variables that store arrangements of values to represent something. For example, in the Tic Tac Toe chapter, a Tic Tac Toe board data structure was a list of strings. The string represented an X, O, or empty space and the index of the string in the list represented the space on the board. The Sonar game will have similar data structures for the locations of treasure chests and sonar devices.

How the Code Works

Lines 3 and 4 import modules and . The module contains the function, which causes the program to terminate immediately. This function is used later in the program.

Drawing the Game Board

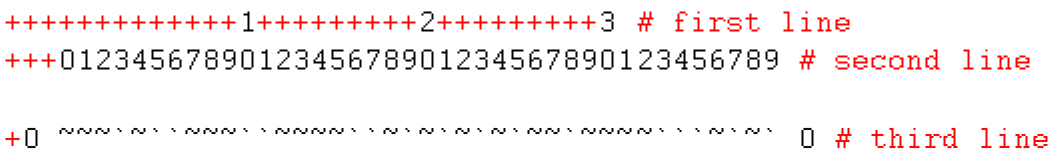
- and Y-axis coordinates around it. The back tick (`) and tilde (~) characters are located next to the 1 key on your keyboard will be used for the ocean waves. It looks like this:

The drawing in the function has four steps.

- First, create a string variable of the line with 1, 2, 3, 4, and 5 spaced out with wide gaps (to mark the coordinates for 10, 20, 30, 40, and 50 on the X-axis).
- Second, use that string to display the X-axis coordinates along the top of the screen.
- Third, print each row of the ocean along with the Y-axis coordinates on both sides of the screen.
- Fourth, print the X-axis again at the bottom. Coordinates on all sides makes it easier to see coordinates for where to place a sonar device.

Drawing the X-Coordinates Along the Top

Look again at the top part of the board in Figure 13-3. It has + plus signs instead of blank spaces so you can count the blank spaces easier:



```
+++++++++++++1+++++++++2+++++++++3 # first line
+++0123456789012345678901234567890123456789 # second line

+0 ~~~`~``~~~``~~~~``~`~`~`~`~~`~~~~```~`~` 0 # third line
```

Figure 13-3: The spacing used for printing the top of the game board.

The numbers on the first line which mark the tens position all have nine spaces between them, and there are thirteen spaces in front of the 1. Lines 9 to 11 create this string with this line and store it in a variable named .

To print the numbers across the top of the sonar board, first print the contents of the variable. Then on the next line, print three spaces (so that this row lines up correctly), and then print the string . But as a shortcut you can use , which evaluates to the same string.

Drawing the Rows of the Ocean

Lines 19 to 25 print each row of ocean waves, including the numbers down the side to label the Y-axis. The loop prints rows 0 through 14, along with the row numbers on either side of the board.

small problem. Numbers with only one digit (like 0, 1, 2, and so on) only take up one space when printed, but numbers with two digits (like 10, 11, and 12) take up two spaces. The e this:

The solution is easy. Add a space only in front of all the single-digit numbers. Lines 21 to 24 set the variable to either a space or an empty string. The variable is always printed, but only has a space character in it for single-digit row numbers. Otherwise, it is the empty string. This way, all of the rows will line up when you print them.

The
waves. Its two parameters are the board data structure stored in the variable and a row

Drawing the X-Coordinates Along the Bottom

Lines 27 to 30 are similar to lines 13 to 16. They print the X-axis coordinates at the bottom of the screen.

Getting the State of a Row in the Ocean

While the " function creates a string for a single row.

First set to the blank string. The Y-axis coordinate is passed as the parameter. The string is made by concatenating , , , and so on up to . This is because the row contains 60 characters, from index to index .

The loop on line 36 iterates over integers to . On each iteration, the next character in the board data structure is copied on to the end of . By the time the loop is done,

Creating a New Game Board

A new data structure is needed at the start of each new game. The data structure is a list of lists of strings. The first list represents the X coordinate. Since the characters across, this first list needs to contain 60 lists. Create a loop that will append 60 blank lists to it.

But is more than just a list of 60 blank lists. Each of the 60 lists represents an X coordinate of the game board. There are 15 rows in the board, so each of these 60 lists must have 15 characters in them. Line 45 is another loop to add 15 single-character strings that represent the ocean.

randomly chosen and strings. If the return value of is , add the string. Otherwise add the string. This will give the ocean a random, choppy look to it.

Remember that the variable is a list of 60 lists, each list having 15 strings. That means to get the string at coordinate 26, 12, you would access , and not . The X coordinate is first, then the Y coordinate.

Finally, the function returns the value in the variable.

Creating the Random Treasure Chests

The game also randomly decides where the hidden treasure chests are. The treasure chests are represented as a list of lists of two integers. These two integers will be the X and Y coordinates of a single chest.

For example, if the chest data structure was , then this would mean there are three treasure chests, one at 2, 2, another chest at 2, 4, and a third one at 10, 0.

The loop will iterate number of times, and on each iteration line 57 appends a list of two random integers. The X coordinate can be anywhere from 0 to 59, and the Y coordinate can be from anywhere between 0 and 14. The expression that is passed to the append method will evaluate to a list value like or or . This list value is appended to .

Determining if a Move is Valid

When the player types in X and Y coordinates of where they want to drop a sonar device, they may not type invalid coordinates. The X coordinate must be between 0 and 59 and the Y coordinate must be between 0 and 14.

The function uses a simple expression that uses operators to ensure that each part of the condition is . If even one part is , then the entire expression evaluates to . This function returns this Boolean value.

Placing a Move on the Board

In the Sonar game, the game board is updated to display a number for each sonar device dropped to show how far away the closest treasure chest is. So when the player makes a move by giving the program an X and Y coordinate, the board changes based on the positions of the treasure chests.

The function takes four parameters: the game board data structure, the treasure chests data structure, and the X and Y coordinates. Line 69 returns if the X and Y coordinates if was passed do not exist on the game board. If returns , then will itself return .

Otherwise, will return a string value describing what happened in response to the move:

- If the coordinates land directly on the treasure, returns .
- If the coordinates are within a distance of 9 or less, returns (where is replaced with the integer distance).
- Otherwise, will return .

Given the coordinates of where the player wants to drop the sonar device and a list of XY "

closest.

An Algorithm for Finding the Closest Treasure Chest

The and parameters are integers (say, and), and together they represent the location on the game board where the player guessed. The variable will have a value such as . That value represents the locations of three treasure chests. You can visualize it as the picture in Figure 13-3. " as in Figure 13-4.

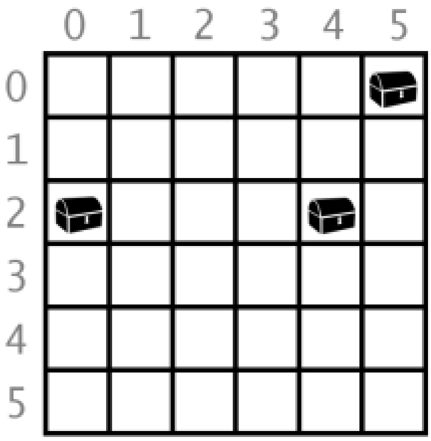
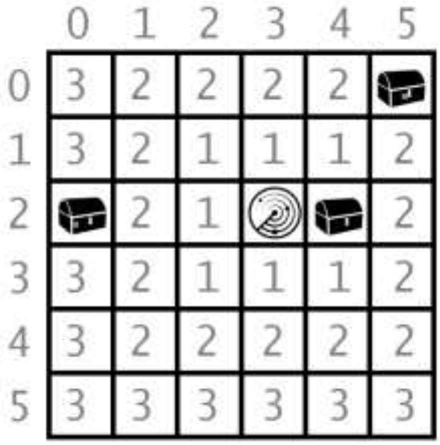


Figure 13-3: The treasure chests that represents.



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

Figure 13-4: The board marked with distances from the 3, 2 position.

But how do you translate this into code for the game? You need a way to represent the square ring distance as an expression. Notice that the distance from an XY coordinate is always the larger of two values: the absolute value of the difference of the two X coordinates and the absolute value of the difference of the two Y coordinates.

coordinate, and then take the absolute value of this number. Do the same for *larger* of these two values is the distance.

" , like in Figure 13-4. The first ") are and .

1. For the X coordinates, evaluates to , and the absolute value of is .
2. For the Y coordinates, evaluates to , and the absolute value of is .
3. 1 0 0 1 274.371 0 0 1 98.3 and Comparing the two absolute values

Line 72 uses the multiple assignment trick in a loop. For example, the assignment statement will assign to and to .

Because is a list where each item in the list is itself a list of two integers, the first of these integers is assigned to and the second integer is assigned to . So if has the value , will have the value and will have the value on the first iteration through the loop.

Line 73 determines which is larger: the absolute value of the difference of the X coordinates, or the absolute value of the difference of the Y coordinates. seems like much shorter " variable.

So on each iteration of the loop, the
from the sonar device. But you want the smallest distance of all the treasure chests. This is where the variable comes in. Whenever the variable is smaller than , then the value in becomes the new value of .

Give the impossibly high value of at the beginning of the loop so that at least one of the treasure chests you found will be put into . By the time the loop has finished, you know that e holds the shortest distance between the sonar device and all of the treasure chests in the game.

The `remove()` List Method

The list method will remove the first occurrence of a value matching the passed in argument. For example, try entering the following into the interactive shell:

The value has been removed from the list. The method removes the first occurrence of the value you pass it, and only the first. For example, type the following into the interactive shell:

Notice that only the first value was removed, but the second and third ones are still there. The method will cause an error if

The only time that is equal to coordinates. This means the player has correctly guessed the location of a treasure chest. R -integer list from the data structure with the list method. Then the function returns .

The else-block starting on line 86 executes if was not , which means the

than 10, line 87 marks the board with the string version of . If not, mark the board with a .

Getting the Player's Move

The loop will keep asking the player for their next move until they enter a valid move. The player can also type in to quit the game. In that case, line 101 calls the function to terminate the program immediately.

Assuming the player has not typed in , the code must ensure it is a valid move: two integers separated by a space. Line 103 calls the method on as the new value of .

If the player typed in a value like , then the list returned by would be . In that case, the expression would be and the entire expression evaluates immediately to because of short-circuiting (which was described in Chapter 10).

then the two values will be at indexes and . To check if those values are numeric digits (like or), you could use a function like from Chapter 11. But Python already has a function that does this.

The string method returns if the string consists solely of numbers. Otherwise it returns . Try entering the following into the interactive shell:

Both and must be for the whole condition to be . the function to check if the XY coordinates exist on the board.

If the entire condition is , line 105 returns a two-integer list of the XY coordinates. Otherwise, the execution loops and the player will be asked to enter coordinates again.

Asking the Player to Play Again

The function is similar to the functions in previous chapters.

Printing the Game Instructions for the Player

The is a couple of calls that print multi-line strings. The function gives the player a chance to press ENTER before printing the next string. This is because the IDLE window can only show so much text at a time.

After the player presses ENTER, the function returns.

The Start of the Game

The expression asks the player if they want to see the instructions, and evaluates to if the player typed in a string that began with or . If so, is called. Otherwise, the game begins.

loop is the main loop for the program. Several variables are set up on lines 173 to 177 and are described in Table 13-1.

Table 13-1: Variables used in the main game loop.

Variable	Description
	The number of sonar devices (and turns) the player has left.
	The board data structure used for this game.
	The list of chest data structures. will return a list of three treasure chests at random places on the board.
	A list of all the XY moves that the player has made in the game.

Displaying the Game Status for the Player

loop executes as long as the player has sonar devices remaining. Line 187 prints a message telling the user how many sonar devices and treasure chests are left. But a small problem.

If there are two or more sonar devices left, you want to print . But if only one sonar device left, you want to print left. You only want the plural

and .

Lines 183 through 186 have code after the and statements' colon. This is perfectly valid Python. Instead of having a block of code after the statement, you can use the rest of the same line to make your code more concise.

The two variables named and are set to (space) if there are multiple sonar devices or treasures chests. Otherwise, they are blank strings. These variables are used on line 187.

Getting the Player's Move

Line 189 uses multiple assignment since returns a two-item list. The first item in the returned list is assigned to the variable. The second is assigned to the variable.

They are then appended to the end of the list. This means is a list of XY coordinates of each move the player makes in this game. This list is used later in the program on line 198.

The , , , and variables are all passed to the function. This function will make the necessary modifications to the game board to place a sonar device on the board.

If returns the value , then there was a problem with the and

Checking if the Player has Lost

Line 207 is the last line of the loop that started on line 179. Decrement the variable because the player has used one. If the player keeps missing the treasure chests, eventually will be reduced to . After this line, execution jumps back up to line 179 so it can re-evaluate the).

If is , then the condition will be and execution will continue outside the while-block on line 209. But until then, the condition will remain and the player can keep making guesses.

Line 209 is the first line outside the loop. When the execution reaches this point the game is over. If is , you know the player ran out of sonar devices before finding all the chests and lost.

loop on line 213 will go through the treasure chests remaining in and show their location to the player so that they can know where the treasure chests had been lurking.

The sys.exit() Function

Win or lose, is called again to let the player type in whether they want to keep playing or not. If not, then returns . The operator on line 216 changes this to , making the and the function is executed. This will cause the program to terminate.

Otherwise, execution jumps back to the beginning of the loop on line 171 and a new game begins.

Summary

Remember how our Tic Tac Toe game numbered the spaces on the Tic Tac Toe board 1 through 9? This sort of coordinate system might have been okay for a board with less than ten spaces. But the Sonar board has 900 spaces! The Cartesian coordinate system we learned in the last chapter really makes all these spaces manageable, especially when our game needs to find the distance between two points on the board.

Locations in games that use a Cartesian coordinate system can be stored in a list of lists so that the first index is the X-coordinate and the second index is the Y-coordinate. This make accessing a coordinates look like .

These data structures (such as the ones used for the ocean and locations of the treasure chests) make it possible to have complicated concepts represented as data, and your game programs become mostly about modifying these data structures.

In the next chapter, we will be representing letters as numbers using their ASCII numbers. (This s, we can perform math 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 and functions
- The string method
- The and string methods
- Cryptanalysis
- The brute force technique

"

normal English into a secret code. It can also convert secret codes back into regular English again. Only someone who is knowledgeable about secret codes will be able to understand our secret messages.

Because this program manipulates text to convert it into secret messages, you will learn several new functions and methods for manipulating strings. You will also learn how programs can do math with text strings just as it can with numbers.

Cryptography

The science of writing secret codes is called **cryptography**. For thousands of years cryptography has made secret messages that only the sender and recipient could read, even if someone captured the messenger and read the coded message. A secret code system is called a **cipher**. The cipher used by the program in this chapter is called the Caesar cipher.

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

Converting the plaintext into the encoded message is called **encrypting** the plaintext. The plaintext is encrypted into the **ciphertext**. The ciphertext looks like random letters, and we cannot understand what the original plaintext was just by looking at the ciphertext. Here is the previous example encrypted into ciphertext:

But if you know about the cipher used to encrypt the message, you can **decrypt** the ciphertext back to the plaintext. (Decryption is the opposite of encryption.)

Many ciphers also use keys. **Keys** are secret values that let you decrypt ciphertext that was encrypted using a specific cipher. Think of the cipher as being like a door lock. You can only unlock it with a particular key.

" "

http://inventwithpython.com/hacking.

The Caesar Cipher

The key for the Caesar Cipher will be a number from 1 to 26. Unless you know the key (that is,

"

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**

" " letter. If you shift the letter A by one space, you get the letter B. If you shift the letter A by two spaces, you get the letter C. Figure 14-1 is a picture of some letters shifted over by three spaces.

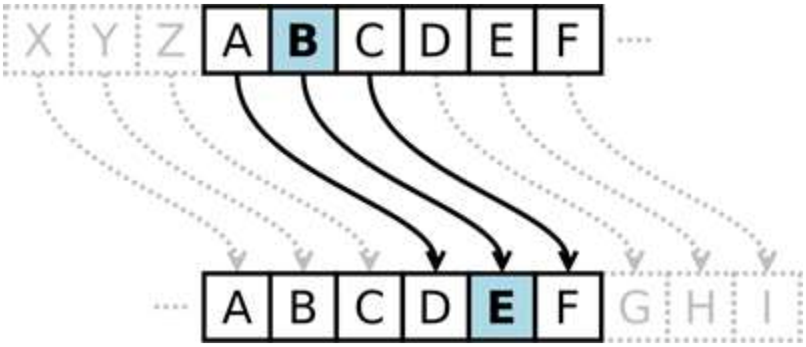


Figure 14-1: Shifting over letters by three spaces. Here, B becomes E.

To get each shifted letter, draw out a row of boxes with each letter of the alphabet. Then draw a second row of boxes under it, but start a certain number (this number is the key) of spaces over. After the letters at the end, wrap around back to the start of the boxes. Here is an example with the letters shifted by three spaces:

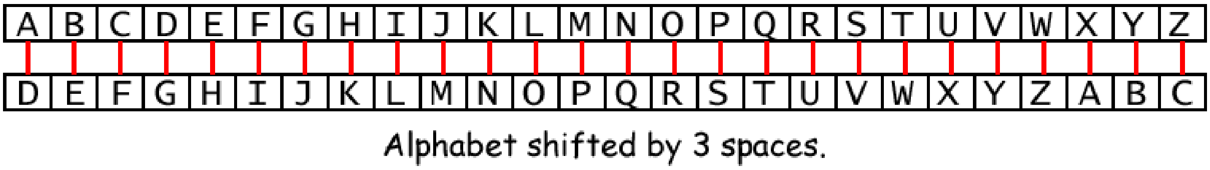

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
Alphabet shifted by 3 spaces.

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

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

If you with a key of 3, then:

- T .
- T .
- T .
- T .
- T

We will keep any non-letter characters the same. T "
the bottom boxes back to the top:

- .
- T .
- T .
- T .
- T .

ASCII, and Using Numbers for Letters

How do we implement this shifting of the letters as code? We can do this by representing each letter as a number called an **ordinal**, and then adding or subtracting from this number to form a new ordinal -
Code for Information Interchange) is a code that connects each character to a number between 32 and 126.

I numbers 65 through 90. The lowercase letters

have the ASCII numbers 48 through 57. Table 14-1 shows all the ASCII characters and ordinals.

Modern computers use UTF-8 instead of ASCII. But UTF-8 is backwards compatible with ASCII, so the UTF-

Table 14-1: The ASCII Table

32	(space)	48	64	80	96	112
33		49	65	81	97	113
34		50	66	82	98	114
35		51	67	83	99	115
36		52	68	84	100	116
37		53	69	85	101	117
38		54	70	86	102	118
39		55	71	87	103	119
40		56	72	88	104	120
41		57	73	89	105	121
42		58	74	90	106	122
43		59	75	91	107	123
44		60	76	92	108	124
45		61	77	93	109	125
46		62	78	94	110	126
47		63	79	95	111	

"

- (65).
- Add 3 to 65, to get 68.
- Convert the ordinal

The and functions can convert between characters and ordinals.

The chr() and ord() Functions

The " ordinal and returns a single-character string. The -character string, and returns the integer ordinal value. Try entering the following into the interactive shell:

On the third line, evaluates to . If you look at the ASCII table, you can see

On the fifth line, evaluates to which evaluates to . The and functions are the opposite of each other.

Sample Run of Caesar Cipher

Here is a sample run of the Caesar Cipher program, encrypting a message:

```
encrypt

The sky above the port was the color of television, tuned to a dead channel.

13
```

Now run the program and decrypt the text that you just encrypted.

```
decrypt

Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb n qrnq punaary.

13

The sky above the port was the color of television, tuned to a dead channel.
```

If you do not decrypt with the correct key, the decrypted text will be garbage data:

```
decrypt
```

```
Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb n qrnq punaary.

15
```

Source Code of Caesar Cipher

Here is the source code for the Caesar Cipher program. After you type this code in, save the file as *cipher.py*. If you get errors after typing this code in, compare the code you typed code with the online diff tool at http://invpy.com/diff/cipher.

How the Code Works

The encryption and decryption processes are the reverse of the other, and even then they still e works.

The first line is just a comment. is a constant that stores the integer in it. reminds us that in this program, the key used in the cipher should be between 1 and 26.

Deciding to Encrypt or Decrypt

The function will let the user type in if they want encryption or decryption mode for the program. The value returned from and is stored in . The condition checks if the string stored in exists in the list returned by
.

This list is , but it is easier for the programmer to type and not type in all those quotes and commas. Use whichever is easiest for you; they both evaluate to the same list value.

This function will return the string in as long as is equal to , ,
, or . Therefore, will return the string or the string (but the user
, " ,

Getting the Message from the Player

The function simply gets the message to encrypt or decrypt from the user and returns it.

Getting the Key from the Player

A valid key here is one that is between the integer values and (remember that will only ever have the value because it is constant). It then returns this key. Line 22 sets to the integer version of what the user typed in, so returns an integer.

Encrypt or Decrypt the Message with the Given Key

does the encrypting and decrypting. It has three parameters:

- sets the function to encryption mode or decryption mode.
- is the plaintext (or ciphertext) to be encrypted (or decrypted).
- is the key that is used in this cipher.

Line 27 checks if the first letter in the variable is the string . If so, then the program is in decryption mode. The only difference between the decryption and encryption mode is that in decryption mode the key is set to the negative version of itself. If was the integer , then in decryption mode set it to . The reason why will be explained later.

is the string of the result: either the ciphertext (if you are encrypting) or the plaintext (if you are decrypting). It starts as the blank string and has encrypted or decrypted characters concatenated to the end of it.

The `isalpha()` String Method

The string method will return if the string is an uppercase or lowercase letter from A to Z. If the string contains any non-letter characters, then will return . Try entering the following into the interactive shell:

Encrypting or Decrypting Each Letter

Line 36 checks if the symbol is an uppercase letter. If so, there are two special cases to worry about. What if was and key was ? If that were the case, the value of here would be the character (The ordinal of is

Check if has a value larger than the ordinal " **subtract** (because there are 26 letters in total) from . After doing this, the value of is . 68 is the correct ordinal value for .

is

If the symbol was not an uppercase or lowercase letter, then line 48 concatenates the original symbol to the string. Therefore, spaces, numbers, punctuation marks, and other

The last line in the function returns the string.

The Start of the Program

The start of the program calls each of the three functions defined previously to get the mode, message, and key from the user. These three values are passed to whose return value (the string) is printed to the user.

Brute Force

"

understand cryptography, it keep a message secret from someone who knows cryptanalysis. While cryptography is the science of making codes, **cryptanalysis** is the science of breaking codes.

```
encrypt

Doubts may not be pleasant, but certainty is absurd.

8

Lwcjba uig vwb jm xtmiaivb, jcb kmzbiqvbg qa ijaczl.
```

The whole point of cryptography is that so if someone else gets their hands on the encrypted message, they cannot figu
the code breaker and all we have is the encrypted text:

Brute force is the technique of trying every possible key until you find the correct one. Because there are only 26 possible keys, it would be easy for a cryptanalyst to write a hacking program than decrypts with every possible key. Then they could look for the key that decrypts to plain

the program.

Adding the Brute Force Mode

First, change lines 7, 9, and 12 (which are in the function) to look like the following (the changes are in bold):

```


                                                      or brute force



                                             brute b



                                                                          or
"brute" or "b"
```

Modify and add the following changes to the main part of the program:

```

    if mode[0] != 'b':



    if mode[0] != 'b':

    else:
        for key in range(1, MAX_KEY_SIZE + 1):
            print(key, getTranslatedMessage('decrypt', message, key))
```

These changes a

call is made and the translated string is printed.

" loop that iterates from all the way up to (which is). Remember that when the function returns a list of integers up to, but not including, the second parameter, which is why you have

. This program will print every possible translation of the message (including the key number used in the translation). Here is a sample run of this modified program:

```
brute

Lwcjba uig vwb jm xtmiaivb, jcb kmzbiqvbg qa ijaczl.






8 Doubts may not be pleasant, but certainty is absurd.
```

" " nglish!

The cryptanalyst can deduce that the original key for this encrypted text must have been . This brute force would have been difficult to do back in the days of Caesars and the Roman Empire, but today we have computers that can quickly go through millions or even billions of keys in a short time.

Summary

Computers are good at doing mathematics. When we create a system to translate some piece of information into numbers (such as we do with text and ordinals or with space and coordinate systems), computer programs can process these numbers quickly and efficiently.

But while our Caesar cipher program here can encrypt messages that will keep them secret from people who have to figure it out with pencil and paper, it keep it secret from people who know how to get computers to process information for them. (Our brute force mode proves this.)

A large part of figuring out how to write a program is figuring out how to represent the information you want to manipulate as values that Python can understand.

The next chapter will present Reversi (also known as Othello). The AI that plays this game will be much more advanced than the AI that played Tic Tac Toe in chapter 9. In fact, the AI is so good

Topics Covered In This Chapter:

- The Function
- How to Play Reversi

In this chapter, we ll make a game called Reversi (also called Othello). Reversi is a board game that is played on a grid, so we ll use a Cartesian coordinate system with XY coordinates. It is a game played with two players. Our version of the game will have a computer AI that is more advanced than the AI we made for Tic Tac Toe. In fact, this AI is so good that it will probably beat you almost every time you play. (I know I lose whenever I play against it!)

Reversi has an 8 × 8 board and tiles that are black on one side and white on the other (our game

d looks like Figure 15-1. The black player and

between the new tile and the other tiles of that color are flipped. The goal of the game is to have as many of the tiles with your color as possible. For example, Figure 15-2 is what it looks like if the white player places a new white tile on space 5, 6.

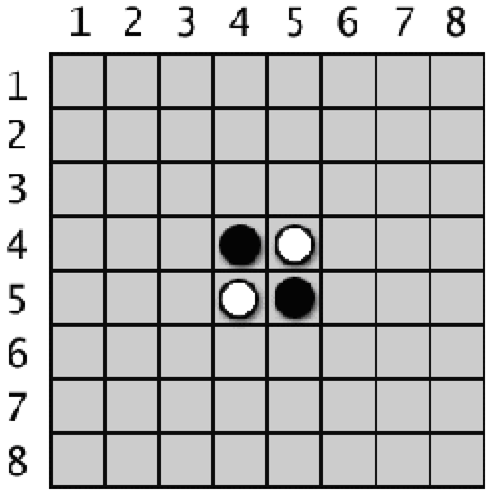


Figure 15-1: The starting Reversi board has two white tiles and two black tiles.

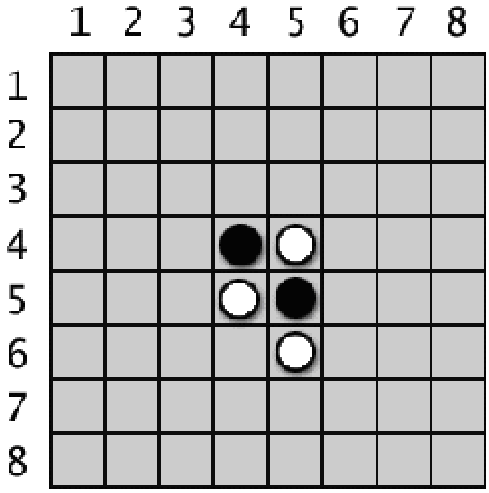


Figure 15-2: White places a new tile.

Each player can quickly flip many tiles on the board in one or two moves. Players must always make a move that captures at least one tile. The game ends when a player either cannot make a move, or the board is completely full. The player with the most tiles of their color wins.

The AI we make for this game will simply look for any corner moves they can take. If there are no corner moves available, then the computer will select the move that claims the most tiles.

Sample Run of Reversi

```
86



no
```

As you can see, the AI was pretty good at beating me 46 to 15. To help the player out, program the game to provide hints. If the player types as their move, they can toggle the hints mode on and off. When hints mode is on, all the possible moves the player can make will show up on the board as characters, like this:

Source Code of Reversi

t

"

more readable.

As with our other programs, we ll first create several functions to carry out Reversi-related tasks that the main section will call. Roughly the first 250 lines of code are for these helper functions, and the last 50 lines of code implement the Reversi game itself.

If you get errors after typing this code in, compare the code you typed
online diff tool at http://invpy.com/diff/reversi.

How the Code Works

The Game Board Data Structure

Before getting into the code,
of lists, just like the one in the previous Sonar game. The list of lists is created so that
will represent the character on space located at position on the X-axis (going left/right) and position on the Y-axis (going up/down).

This character can either be a space character (to represent a blank space), a period character (to represent a possible move in hint mode), or an or
tile). Whenever you see a parameter named , it is meant to be this kind of list of lists data structure.

Importing Other Modules

The function will print the current game board based on the data structure in . Notice that each square of the board looks like this (there could also be a , , or string instead of the):

Since the horizontal line is printed over and over again, line 8 stores it in a constant variable named . This will save you from typing out the string repeatedly.

There are also lines above and below the center of tile that are nothing but characters (called .

Line 11 is the first function call executed, and it prints the labels for the X-axis along the top of the board. Line 12 prints the top horizontal line of the board.

The loop will loop eight times, once for each row. Line 15 prints the label for the Y-axis on the left side of the board, and has an keyword argument to print a single space instead of a new line. This is so that another loop (which again loops eight times, once for each space) prints each space (along with the , , or character depending on what is stored in .)

The function call inside the inner loop also has an keyword argument at the end of it, meaning a space character is printed instead of a newline character. That will produce a single line on the screen that looks like (if each of the values were).

After the inner loop is done, the function call on line 18 prints the final character along with a newline.

The code inside the outer loop from line 14 to line 20 prints an entire row of the board like this:

When the loop on line 13 prints the row eight times, it forms the entire board (of course, some of the spaces on the board will have or instead of):

Resetting the Game Board

Line 25 and 26 have nested loops to set the data structure to be all single-space strings. This makes a blank Reversi board. The function is called as part of starting a new game.

Setting Up the Starting Pieces

At the beginning of a game, each player has two tiles already laid down in the center. Lines 30 to 33 set those tiles on the blank board.

The function does not have to return the variable, because is a modify the original list that was passed as the argument. (See the References section in Chapter 10.)

Creating a New Game Board Data Structure

The function creates a new board data structure and returns it. Line 38 creates the outer list and stores a reference to this list in board. Line 40 creates the inner lists using list replication. (evaluates to be the same as but with less typing.)

loop here creates the eight inner lists. The spaces represent a completely empty game board.

What ends up being is a list of eight lists, and each of those eight lists themselves has eight strings. The result is 64 strings.

Checking if a Move is Valid

" " "

should return if the Reversi game rules allow a move to those coordinates and

Line 48 checks if the XY coordinates are not " is a function defined later in the program that makes sure both the X and Y coordinates are between and .

setting the board space back to before returning).

the human player or the computer player) is in , but this function then obviously the other , and vice versa.

Finally, if the given XY coordinate ends up as a valid position, returns a list of all the opponent's tiles that would be flipped by this move.

The loop iterates through a list of lists which represent directions you can move on the game board. The game board is a Cartesian coordinate system with an X and Y direction. There are eight directions you can move: up, down, left, right, and the four diagonal directions. Each of the eight 2-item lists in the list on line 59 is used for moving in one of these directions. The program moves in a direction by adding the first value in the two-item list to the X coordinate, and the second value to the Y coordinate.

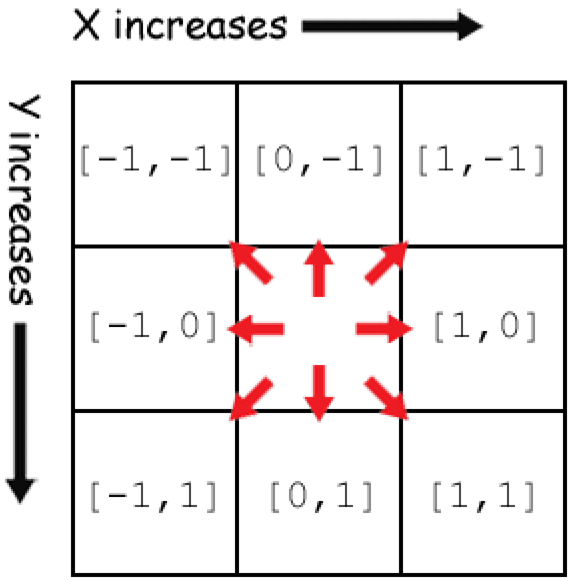
Because the X coordinates i " to the X coordinate. So the list adds to the X coordinate and to the Y coordinate,

(that is, add) from the X coordinate.

But to move diagonally, you need to add or subtract to both coordinates. For example, adding to the X coordinate to move right and adding to the Y coordinate to move up would result in moving to the up-right diagonal direction.

Checking Each of the Eight Directions

Here is a diagram to make it easier to remember which two-item list represents which direction:



[-1,-1]	[0,-1]	[1,-1]
[-1,0]		[1,0]
[-1,1]	[0,1]	[1,1]

Figure 15-7: Each two-item list represents one of the eight directions.

Line 60 sets an and variable to be the same value as and , respectively, using multiple assignment. Change and at and dictate. The and variables will stay the same so that the program can remember which space it originally started from.

Remember, in order for this to be a valid move, the first step in this direction must be 1) on the

opponent "

" and the execution goes back to the statement for the next direction.

"

though, then line 68 should continue back to the statement to try the next direction.

The loop on line 69 keeps looping so that and keep going in the current direction as long as it keeps f line 72 detects that and moved off of the board, line 73 breaks out of the loop and the flow of execution moves to line 74.

What you really want to do is break out of the loop but continue in the loop. This is why line 74 rechecks and runs , which moves execution to the statement. Remember, and statements will only break or continue from the innermost loop they are in.

Finding Out if There are Pieces to Flip Over

loop stops looping when the code has reached the end of the tiles. Line 76 checks if this space on the board holds one of our tiles. If it does, then the move originally passed to is valid.

Line 78 loops by moving and in reverse back to the original and position by subtracting and . Each space is appended to the list.

The loop that started on line 59 does this in all eight directions. After that loop is done, the

if the player moved on , . Remember, the function is only checking to see if the original move was valid. I permanently change the data structure of the game board.

If none of the eight directions ended up flipping "
will be an empty list. This is a sign that this move is not valid and should return .

Otherwise, returns .

Checking for Valid Coordinates

is a function called from . Calling the function is shorthand for the Boolean expression on line 93 that is if both and are between and . This function checks if an X and Y coordinate is actually on the game board. For example, an X coordinate of and a Y coordinate of would not be on the board since Y coordinates only go up to .

Getting a List with All Valid Moves

returns a game board data structure that has characters for all spaces that are valid moves. The periods are for the hints mode that displays a board with all possible moves marked on it.

This function creates a duplicate game data structure (returned by on line 98) instead of modifying the one passed to it in the parameter. Line 100 calls to get a list of XY coordinates with all the legal moves the player could make. The board copy is marked with periods in those spaces and returned.

The function returns a list of two-item lists. These lists hold the XY coordinates for all valid moves for parameter.

This function uses nested loops (on lines 109 and 110) to check every XY coordinate (all sixty four of them) by calling on that space and checking if it returns or a list of possible moves (in which case it is a valid move). Each valid XY coordinate is appended to the list in .

The bool() Function

The is similar to the and functions. It returns the Boolean value form of the value passed to it.

Most data types have one value that is considered the value for that data type. Every other value is consider . For example, the integer , the floating point number , the empty string, the empty list, and the empty dictionary are all considered to be when used as the condition for an or loop statement. All other values are . Try entering the following into the interactive shell:

. That is, conditions are automatically interpreted as Boolean values. This is why the condition on line 111 works correctly. The call to the function either returns the Boolean value or a non-empty list.

If you imagine that the entire condition is placed inside a call to , then condition becomes (which, of course, evaluates to). And a condition of a non-empty list placed as the parameter to will return .

Getting the Score of the Game Board

The function uses nested loops to check all 64 spaces on the board (8 rows times 8 columns per row is 64 spaces) and see which tile (if any) is on them. For each tile, the code increments on line 123. For each tile, the code increments on line 125

Getting the Player's Tile Choice

This function asks the player which tile they want to be, either or . The loop will keep looping until the player types in or .

The function then returns a two- "
Line 252, which calls ,
uses multiple assignment to put these two returned items in two variables.

Determining Who Goes First

The function randomly selects who goes first, and returns either the string or the string .

Asking the Player to Play Again

The function was also in previous games. If the player types in a string that begins with , then the function returns . Otherwise the function returns .

Placing Down a Tile on the Game Board

is called when you want to place a tile on the board and flip the other tiles according to the rules of Reversi. This function modifies the data structure that is passed in-place. Changes made to the variable (because it is a list reference) will be made to the global scope.

Most of the work is done by , which returns a list of XY coordinates (in a two-item list) of tiles that need to be flipped. (Remember, if the and arguments point to an invalid move, then will return the Boolean value .)

On lines 163 and 164, if the return value of (now stored in) was , then will also return .

Otherwise, returns a list of spaces on the board to put down the tiles (the or string in). Line 166 sets the space that the player has moved on. loop sets all the tiles that are in .

Copying the Board Data Structure

is different from . will create a blank game board data structure which has only empty spaces and the four starting tiles. will create a blank game board data structure, but then copy all of the spaces from the parameter. This function is used by the AI to have a game board that it can change around without changing the real game board. This technique was also used by the previous Tic Tac Toe program.

A call to handles getting a fresh game board data structure. Then the two nested loops copy each of the 64 tiles from board to the duplicate board data structure in .

Determining if a Space is on a Corner

The function returns if the coordinates are on a corner space at coordinates (0,0), (7,0), (0,7) or (7,7). Otherwise returns .

Getting the Player's Move

The 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 to turn hints mode on (if it is off) or off (if it is on). The player can also type in to quit the game.

The constant variable is the list . The constant is used because it is easier to type than the entire list. You method because that would allow 0 and 9 to be entered, which are not valid coordinates on the 8 × 8 board.

The loop will keep looping until the player has typed in a valid move. Lines 195 to 198 check if the player wants to quit or toggle hints mode, and return the string or , respectively. The method is called on the string returned by so the player can type or but still have the command understood.

The code that called will handle what to do if the player wants to quit or toggle hints mode.

The game is expecting that the player would have typed in the XY coordinates of their move as two numbers without anything between them. Line 200 first checks that the size of the string the player typed in is . After that, it also checks that both (the first character in the string) and (the second character in the string) are strings that exist in .

Remember that the game board data structures have indexes from 0 to 7, not 1 to 8. The code prints 1 to 8 when the board is displayed in because non-programmers are used to numbers beginning at 1 instead of 0. So to convert the strings in and to integers, lines 201 and 202 subtract 1.

Even if the player typed in a correct move, the code still needs to check that the move is allowed by the rules of Reversi. This is done by which is passed the game board data " "

If returns , statement executes. The execution will then go back to the beginning of the loop and asks the player for a valid move again.

Otherwise, the player did type in a valid move and the execution needs to break out of the loop.

If the , t
Lines 208 and 209 instructs them on how to correctly type in moves. Afterwards, the execution moves back to the statement -
block, but also the last line in the while-block.

Finally, returns a two-
move.

Getting the Computer's Move

and is where the AI algorithm is implemented. Normally you use the results from for hints mode. Hints mode will print period characters on the board to show the player all the potential moves they can make.

But if), it will also find all the possible moves that the computer can make. The AI will select the best move from this list.

First, will randomize the order of moves in the list. Why we want to shuffle the list will be explained later"

Corner Moves are the Best Moves

First, line 223 loops through every move in . If any of them are on the corner, return that space is returned as the move. Corner moves are a good idea in Reversi because once a tile has been placed on the corner, it can never be flipped over. Since is a list of two-item lists, use multiple assignment in the loop to set and .

If contains multiple corner moves, the first one is always used. But since was shuffled on line 220, it is random which corner move is first in the list.

Get a List of the Best Scoring Moves

If there are no corner moves, loop through the entire list and find out which move results in the loop will set and to every move in . is

set to the highest scoring move the code has found so far, and
score.

When the code in the loop finds a move that scores higher than , line 233 to 235 will store that move and score as the new values in and .

Simulate All Possible Moves on Duplicate Board Data Structures

Before simulating a move, line 230 makes a duplicate game board data structure by calling
board
data structure stored in the variable.

Then line 231 calls , passing the duplicate board (stored in) instead of the real board. This will simulate what would happen on the real board if this move was made.
will handle pla
duplicate board.

Line 232 calls with the duplicate board, which returns a dictionary where the keys are and , and the values are the scores.

For example, pretend that returns the dictionary and
is . Then would evaluate to
, which would then evaluate to . If is larger than ,
is set to and is set to the current and values.

By the time this loop is finished, you can be sure that is the highest possible score a move can make, and that move is stored in .

Line 228 first sets to so that the first move the code checks will be set to the first
. This will guarantee that is set to one of the moves from
when it returns.

" random because
the list order was shuffled on line 220. This ensures that the AI be predictable when
more than one best move.

Printing the Scores to the Screen

calls the function and then prints the s scores. Remember that returns a dictionary with the keys and and values of the scores for the X and O players.

actual game and calls these functions as needed.

The Start of the Game

The loop on line 248 is the main game loop. The program will loop back to line 248 when a new game starts. First get a new game board data structure by calling and set the starting tiles by calling . is the main game board data structure for the program. The call to will let the player type in whether they want to be or . The return value is then stored in and using multiple assignment.

is a Boolean value that determines if hints mode is on or off. It starts as off as on line 253.

The variable is a string that either has the string value or . It will keep track of whose turn it is. It is set to the return value of , which randomly chooses who will go first.

Running the Player’s Turn

The loop that starts on line 257 will keep looping each time the player or computer takes a turn. The execution will break out of this loop when the current game is over.

Line 258 has an statement
else-

First the board is displayed on the screen. If hints mode is on (that is, is), then the board data structure needs to have period characters on every valid space the player could move.

The function does that. It is passed a game board data structure and returns a copy that also contains period characters. Line 262 passes this board to the function.

If hints mode is off, then line 264 passes to .

After printing out the game board to the player, you also want to print the current score by calling on line 265.

Next, let the player type in their move. handles this, and its return value is a two- has already made sure that the move the player typed in is a valid move.

Handling the Quit or Hints Commands

If the player typed in the string for their move, then would have returned the string . In that case, line 269 calls the to terminate the program.

If the player typed in the string for their move, then would have returned the string . In that case, you want to turn hints mode on (if it was off) or off (if it was on).

The assignment statement on line 271 handles both of these cases, because evaluates to and evaluates to . Then the statement moves the execution to the start of the loop (has not changed, so it will still be the

" " to make the

Make the Player's Move

" to see if the computer could make any moves. If returns a blank list, then there are no more valid moves that the computer could make. In that case, line 277 breaks out of the loop and ends the game.

Otherwise, line 279 sets to . The flow of execution skips the else-block and reaches the end of the while-block, so execution jumps back to the statement on line 257.
" "

Running the Computer's Turn

After printing out the board with , also print the current score with a call to on line 284.

Line 285 calls to pause the script while the player can look at the board. This is much like how was used to pause the program in the Jokes chapter. Instead of using a call to print a string before a call to , you can do the same thing by passing the string to print to .

After the player has looked at the board and pressed ENTER, line 286 calls to are stored in variables and using multiple assignment.

Finally, pass and " " to reflect call to stored it in variables and). The call to on line 287 makes the move on the board.

Lines 289 to 292 are similar to lines 276 to 279. After the computer has made its move, line 289 checks if there exist any valid moves the human player can make. If returns an empty list, then there are no valid moves. That means the game is over, and line 290 breaks out of the loop.

Otherwise, at least one possible move the player should make. The variable is to . There is no more code in the while-block after line 292, so execution loops back to the statement on line 257.

Drawing Everything on the Screen

Line 294 is the first line beyond the while-block that started on line 257. This code is executed when the execution breaks out of that loop from line 290 or 277. At this point, the game is over. Now the program should print the board and scores and determine who won the game.

will return a dictionary with keys and and values of both players
" "
score, you can know if the player won, lost, or tied, respectively.

Ask the Player to Play Again

Call the function, which returns if the player typed in that they want to play another game. If returns , the operator makes the condition , the execution breaks out of the loop that started on line 248. Since there are no more lines of code after this while-block, the program terminates.

Otherwise, has returned (making the), and so execution loops back to the statement on line 248 and a new game board is created.

Changing the drawBoard() Function

The board you draw for the Reversi game is large. But you could change the
" he
same. The new, smaller board would look like this:

Here is the code for this new function, starting at line 6. You can also download this code from http://invpy.com/reversi_mini.py.

Summary

"

follows is simple: move on the corner if you can, otherwise make the move that will flip over the most tiles. We could do that, but it would be slow to figure out how many tiles would be flipped for every possible valid move we could make. But calculating this for the computer is simple. The " ter!

This game is similar to Sonar because it makes use of a grid for a board. It is also like the Tic Tac Toe game because an AI that plans out the best move for it to take. This chapter only introduced one new concept: that empty lists, blank strings, and the integer all evaluate to in the context of a condition.

"

know much about programming to create interesting games. However, this game is stretching how far you can get with ASCII art. The board took up almost the entire screen to draw, and the game didn't have any color.

Later in this book, we will learn how to create games with graphics and animation, not just text. We will do this using a module called Pygame, which adds new functions and features to Python so that we can break away from using only text and keyboard input.

Topics Covered In This Chapter:

- Simulations
- Percentages
- Pie Charts
- Integer Division
- The function
- Computer vs. Computer Games

The Reversi AI algorithm was simple, but it beats me almost every time I play it. This is because the computer can process instructions fast, so checking each possible position on the board and selecting the highest scoring move is easy for the computer. It would take a long time for me to find the best move this way.

The Reversi program in Chapter 14 had two functions, and which both returned the move selected as a two-item list like The both also had the same parameters, the game board data structure and which tile they were. decided which move to return by letting the player type in the coordinates. decided which move to return by running the Reversi AI algorithm.

What happens when we replace the call to with a call to ? Then the player never types in a move, it is decided for them! The computer is playing against itself!

We will make three new programs, each based on the Reversi program in the last chapter:

- *AISim1.py* will be made by making changes to *reversi.py*
- *AISim2.py* will be made by making changes to *AISim1.py*
- *AISim3.py* will be made by making changes to *AISim2.py*

You can either type these changes in yourself, or download them from the website at the URL http://invpy.com/chap16.

Making the Computer Play Against Itself

Save the old *reversi.py* file as *AISim1.py* by clicking on **File ► Save As**, and then entering *AISim1.py* for the file name and clicking Ok. This will create a copy of our Reversi source code as a new file that you can make changes to, while leaving the original Reversi game the same (you may want to play it again). Change the following code in *AISim1.py*:

To this (the change is in **bold**):

```
              getComputerMove
```

Now run the program. Notice that the game still asks you if you want to be X or O, but it ask you to enter any moves. When you replaced , you no longer call any code that takes this input from the player. You still press ENTER after the original computer s moves (because of the) on line 285), but the game plays itself!

make some other changes to *AISim1.py*. All of the functions you defined for Reversi can stay the same. But replace the entire main section of the program (line 246 and on) to look like the following code. Some of the code has remained, but most of it has been altered. But all of the lines before line 246 are the same as in Reversi in the last chapter. You can also avoid typing in the code by downloading the source from the URL http://invpy.com/chap16.

If you get errors after typing this code in, compare the code you typed to the code with the online diff tool at http://invpy.com/diff/AISim1.

How the AISim1.py Code Works

The *AISim1.py* program is the same as the original Reversi program, except that the call to has been replaced with a call to . There have been some other changes to the text that is printed to the screen to make the game easier to follow.

When you run the *AISim1.py* program, all you can do is press Enter for each turn until the game ends. Run through a few games and watch the computer play itself. Since both the X and O players are using the same algorithm, it really is just a matter of luck to see who wins. The X player will win half the time, and the O player will win half the time.

Making the Computer Play Itself Several Times

But what if we created a new algorithm? Then we could set this new AI against the one implemented in , and see which one is better. make some changes to the source code. Do the following to make *AISim2.py*:

1. Click on **File ► Save As**.
2. Save this file as *AISim2.py* so that you can make changes without affecting *AISim1.py*. (At this point, *AISim1.py* and *AISim2.py* will have the same code.)
3. Make changes to *AISim2.py* and save that file. (*AISim2.py* will have the new changes and *AISim1.py* will have the original, unchanged code.)

Add the following code. The additions are in bold, and some lines have been removed. When you are done changing the file, save it as *AISim2.py*.

If this is confusing, you can always download the *AISim2.py* source code from the website at http://invpy.com/chap16.

AISim2.py

If you get errors after typing this code in, compare the code you typed to the code with the online diff tool at http://invpy.com/diff/AISim2.

How the AISim2.py Code Works

You have added the variables , , and to lines 248 to 250 to keep track of how many times X wins, O wins, and when they tie. Lines 284 to 289 increment these variables at the end of each game, before it loops back to start a new game.

You have removed most of the function calls from the program, as well as the calls to . When you run *AISim2.py*, it asks you how many games you want to run. Now that you taken out the call to and replace the loop with a loop, you can run a number of games without stopping for the user to type anything. Here is a sample run of ten of computer vs. computer Reversi games:

Because the algorithms include randomness, your run have the exact numbers as above.

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

Percentages

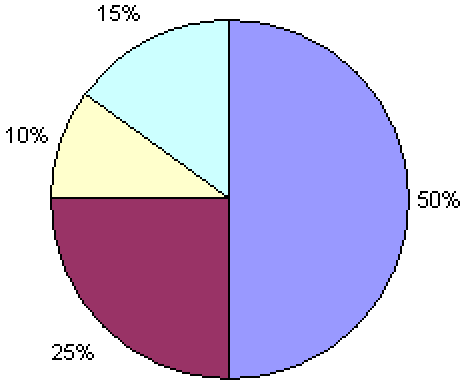


Figure 16-1: A pie chart with 10%, 15%, 25%, and 50% portions.

Percentages are a portion of a total amount, and range from 0% to 100%. If you had 100% of a pie, you would have the entire pie. If you had 0% of a pie, you wouldn't have any pie at all. 50% of the pie would be half of the pie. A pie is a common image to use for percentages. In fact, a kind of chart called a **pie chart** which shows how much of the full total a certain portion is. Figure 16-1 is a pie chart with 10%, 15%, 25%, and 50% portions below. Notice that 10% + 15% + 25% + 50% adds up to 100%: a whole pie.

We can calculate the percentage with division. To get a percentage, divide the part you have by the total, and then multiply by one hundred. For example, if X won 50 out of 100 games, you would calculate the expression , which would evaluate to . Multiply this by to get a percentage (in this case, 50%).

Notice that if X won 100 out of 200 games, you could calculate the percentage with , which would also evaluate to . When you multiply by to get the percentage, you get 50%. Winning 100 out of 200 games is the same percentage (that is, the same portion) as winning 50 out of 100 games.

Division Evaluates to Floating Point

It is important to note that when you use the division operator, the expression will always evaluate to a floating point number. For example, the expression will evaluate to the floating point value , not to the integer value .

This is important to remember, because adding an integer to a floating point value with the addition operator will also always evaluate to a floating point value. For example, will evaluate to the floating point value and not to the integer .

Try entering the following code into the interactive shell:

Notice that in the above example, the data type of the value stored in is always a floating point value. You can pass the floating point value to the function, which will return an integer form of the floating point value. But this will always round the floating point value down. For example, the expressions , , and will all evaluate to , and never .

The round() function

The function will round a float number to the nearest whole float number. Try entering the following into the interactive shell:

The function also has an optional parameter, where you can specify to what place you want to round the number to. For example, the expression evaluates to and evaluates to .

Displaying the Statistics

The code at the bottom of the program will show the user how many wins X and O had, how many ties there were, and how what percentages these make up. Statistically, the more games you run, the more accurate your percentages will be for finding the best AI algorithm. If you only ran ten games, and X won three of them, then it would seem that X s algorithm only wins 30% of the time. However, if you run a hundred, or even a thousand games, then you may find that X s algorithm wins closer to 50% (that is, half) of the games.

To find the percentages, divide the number of wins or ties by the total number of games. Then multiply the result by . However, you may end up with a number like . So pass this number to the function with the second parameter of to limit the precision to two decimal places, so it will return a float like instead (which is much more readable).

try another experiment. Run *AISim2.py* again, but this time have it run a hundred games:

Sample Run of AISim2.py

Depending on how fast your computer is, this run might have taken a about a couple minutes. You can see that the results of all one hundred games still evens out to about fifty-fifty, because both X and O are using the same algorithm to win.

Comparing Different AI Algorithms

add some new functions with new algorithms. But first click on **File ► Save As**, and save this file as *AISim3.py*. Before the line, add these functions in the following source code listing.

AISim3.py

If you get errors after typing this code in, compare the code you typed to the code with the online diff tool at http://invpy.com/diff/AISim3.

How the AISim3.py Code Works

A lot of these functions are similar to one another, and some of them use the new function. a review of the new algorithms made:

Table 17-1: Functions used for our Reversi AI.

Function	Description
	Randomly choose a valid move to make.
	Take a corner move if available. If no corner, take a space on the side. If no sides are available, use the regular algorithm.
	Take a side space if one available. If not, then use the regular algorithm. This means side spaces are chosen before corner spaces.
	Take the space that will result in the fewest tiles being flipped.
	Take a corner space, if available. If not, use the algorithm.

Comparing the Random Algorithm Against the Regular Algorithm

Now the only thing to do is replace one of the calls in the main part of the program with one of the new functions. Then you can run several games and see how often one algorithm wins over the other. First, replace O s algorithm with the one in with on line 351:

When you run the program with a hundred games now, it will look something like this:

```
...skipped for brevity...
```

Wow! X won far more often than O did. That means that the algorithm in (take any available corners, otherwise take the space that flips the most tiles) wins more games than the algorithm in (which makes moves randomly). This makes sense, because making intelligent choices is usually better than just choosing things at random.

Comparing the Random Algorithm Against Itself

s algorithm to also use the algorithm in ? find out s function call on line 351 from to and running the program again.

```
...skipped for brevity...
```

As you can see, when both players are making random moves, they each win about 50% of the time. (In the above case, O happen to get lucky and won a little bit more than half of the time.)

Just like moving on the corner spaces is a good idea because they cannot be flipped, moving on the side spaces may also be a good idea. On the side, the tile has the edge of the board and as out in the open as the other pieces. The corners are still preferable to the side spaces, but moving on the sides (even when a move that can flip more pieces) may be a good strategy.

Comparing the Regular Algorithm Against the CornersSideBest Algorithm

s algorithm on line 346 to use (the original algorithm) and O s algorithm on line 351 to use (which first tries to move on a corner, then tries to move on a side space, and then takes the best remaining move), and run a hundred games to see which is better. Try changing the function calls and running the program again.

```
...skipped for brevity...
```

Wow! unexpected. It seems that choosing the side spaces over a space that flips more tiles is a bad strategy to use. The benefit of the side space greater than the cost of flipping fewer of the opponent s tiles. Can we be sure of these results? run the program again, but this time play one thousand games. This may take a few minutes for your computer to run (but it would take weeks for you to do this by hand!) Try changing the function calls and running the program again.

```
...skipped for brevity...
```

The more accurate statistics from the thousand-games run are about the same as the statistics from the hundred-games run. It seems that choosing the move that flips the most tiles is a better idea than choosing a side move.

Comparing the Regular Algorithm Against the Worst Algorithm

Now set the X algorithm on line 346 to use and the O algorithm on line 351 to (which makes the move that flips over the least number of tiles), and run a hundred games. Try changing the function calls and running the program again.

Whoa! The algorithm in , which always chose the move that flips the fewest tiles, will almost always lose to the regular algorithm. This really surprising at all. (In fact, surprising that this strategy wins even 2% of the time!)

Comparing the Regular Algorithm Against the WorstCorner Algorithm

How about when we replace on line 351 with ? This is the same algorithm except it takes any available corner pieces before taking the worst move. Try changing the function calls and running the program again.

```
...skipped for brevity...
```

The still loses most of the games, but it seems to win a few more games than (6% compared to 2%). Does taking the corner spaces when they are available really make a difference?

Comparing the Worst Algorithm Against the WorstCorner Algorithm

You can check by setting X s algorithm to and O s algorithm to , and then running the program. Try changing the function calls and running the program again.

```
...skipped for brevity...
```

Yes, even when otherwise making the worst move, it does seem like taking the corners results in many more wins. While found out that going for the sides makes you lose more often, going for the corners is always a good idea.

Summary

This chapter didn't really cover a game, but it modeled various strategies for Reversi. If we thought that taking side moves in Reversi was a good idea, we would have to spend weeks, even months, carefully playing games of Reversi by hand and writing down the results. But if we know how to program a computer to play Reversi, then we can have the computer play Reversi using these strategies for us. If you think about it, realize that the computer is executing millions of lines of our Python program in seconds! Your experiments with the simulation of Reversi can help you learn more about playing Reversi in real life.

In fact, this chapter would make a good science fair project. Your problem can be which set of moves leads to the most wins against other sets of moves, and make a hypothesis about which is the best strategy. After running several simulations, you can determine which strategy works best. With programming you can make a science fair project out of a simulation of any board game! And it is all because you know how to instruct the computer to do it, step by step, line by line. You can speak the computer s language, and get it to do large amounts of data processing and number crunching for you.

all for the text-based games in this book. Games that only use text can be fun, even though simple. But most modern games use graphics, sound, and animation to make much more exciting looking games. For the rest of the chapters in this book, we will learn how to create games with graphics by using a Python module called Pygame.

Topics Covered In This Chapter:

- Installing Pygame
- Colors and Fonts in Pygame
- Aliased and Anti-Aliased Graphics
- Attributes
- The , , , and Data Types
- Constructor Functions
-
- The Method for Surface Objects
- Events
- Animation

So far, all of our games have only used text. Text is displayed on the screen as output, and the player types in text from the keyboard as input. Just using text makes programming easy to learn. "

using the Pygame module.

Chapters 17, 18, and 19 teaches you how to use Pygame to make games with graphics, animation, "

The game in Chapter 20 will use all these concepts together to create a game.

Installing Pygame

" free to download. In a web browser, go to the URL http://invpy.org/downloadpygame and download the Pygame installer file for your operating system and version of Python.

Open the installer file after downloading it, and follow the instructions until Pygame has finished installing. To check that Pygame installed correctly, type the following into the interactive shell:

If nothing appears after you hit the ENTER key, then you know Pygame was successfully installed. If the error appears, try to install Pygame again (and make sure you typed correctly).

Figure 17-1: The pygame.org website.

The Pygame website at http://pygame.org has information on how to use Pygame, as well as several other game programs made with Pygame. Figure 17-1 shows the Pygame website.

Hello World in Pygame

The first

"

instead of as text.

of this, you can only write Pygame programs and cannot send instructions to Pygame one at a time through the interactive shell.

Pygame programs also do not use the function. There is no text input and output. Instead, the program displays output in a window by drawing graphics and text to the window.

Events are explained in the next chapter.

Source Code of Hello World

Type in the following code into the file editor, and save it as *pygameHelloWorld.py*. If you get errors after typing this code in, compare the code you typed diff tool at http://invpy.com/diff/pygameHelloWorld.

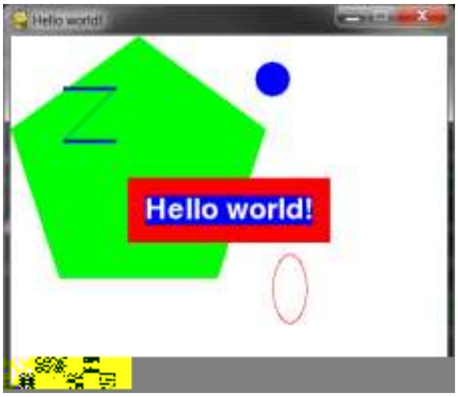


Figure 17-2: The “Hello World” program.

First you need to import the module so you several modules on the same line by delimiting the module names with commas. Line 1 imports both the and modules.

The second line imports the module. This module contains many constant variables that use with Pygame such as or (explained later). However, using the form you can import the module but not have to type in f constants.

If you have instead of in your program, you could call instead of in your code. But most of the time it is better to use the full function name so you know which module the function is in.

The `pygame.init()` Function

All Pygame programs must call the after importing the module but before calling any other Pygame functions. steps.

Tuples

Tuple values are similar to lists, except they use parentheses instead of square brackets. Also, like strings, tuples cannot be modified. For example, try entering the following into the interactive shell:

The `pygame.display.set_mode()` and `pygame.display.set_caption()` Functions

Line 8 creates a GUI window by calling the method in the module. (The module is a module inside the module. Even the module has its own modules!)

A pixel is the tiniest dot on your computer screen. A single pixel on your screen can light up into any color. All the pixels on your screen work together to display all the pictures you see. To create a window 500 pixels wide and 400 pixels high, use the tuple for the first parameter to .

There are three parameters to the method. The first is a tuple of two integers for the width and height of the window, in pixels. The second and third options are advanced options that are beyond the scope of this book. Just pass and for them, respectively.

The function returns a object (which we will call objects for short). **Objects** is just another name for a value of a data type that has methods. For example, strings are objects in Python because they have data (the string itself) and methods (such as and). The object represents the window.

Variables store references to objects just like they store reference for lists and dictionaries. The References section in Chapter 10 explains references.

RGB Colors

Table 17-1: Colors and their RGB values.

Color	RGB Values
Black	
Blue	
Gray	
Green	
Lime	
Purple	
Red	
Teal	
White	
Yellow	

There are three primary colors of light: red, green and blue. By combining different amounts of these three colors (which is what your computer screen does), you can form any other color. In Pygame, tuples of three integers are the data structures that represent a color. These are called **RGB Color** values.

The first value in the tuple is how much red is in the color. A value of means no red in this color, and a value of means a maximum amount of red in the color. The second value is for green and the third value is for blue. These three integers form an RGB tuple.

For example, the tuple has no amount of red, green, or blue. The resulting color is completely black. The tuple has a maximum amount of red, green, and blue, resulting in white.

The tuple represents the maximum amount of red but no amount of green and blue, so the resulting color is red. Similarly, is green and is blue.

You can mix the amount of red, green, and blue to get any shade of any color. Table 17-1 has some common colors and their RGB values. The web page http://invpy.com/colors also lists several more tuple values for different colors.

Fonts, and the `pygame.font.SysFont()` Function

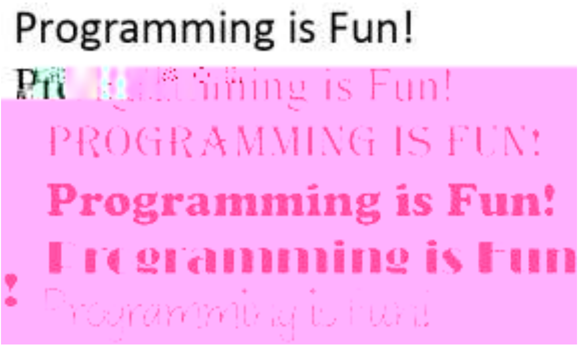


Figure 17-3: Examples of different fonts.

A font is a complete set of letters, numbers, symbols, and characters drawn in a single style. Figure 17-3 shows the same sentence printed in different fonts.

In our earlier games, we only told Python to print text. The color, size, and font that was used to display this text was completely determined by your operating system. The Python program " nt on your computer.

Line 19 creates a object (called Font objects for short) by calling the function. The first parameter is the name of the font, but we will pass the value to use the default system font. The second parameter is the size of the font (which is measured in units called *points*).

The `render()` Method for `Font` Objects

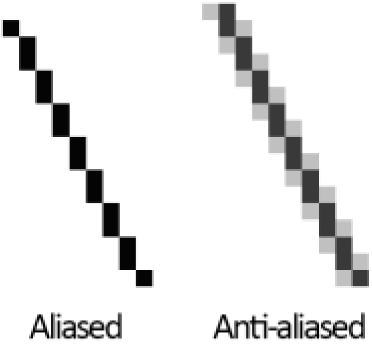


Figure 17-4: An enlarged view of an aliased line and an anti-aliased line.

The object that stored in the variable has a method called . This method will return a object with the text drawn on it. The first parameter to is the string of the text to draw. The second parameter is a Boolean for whether or not you want anti-aliasing.

On line 22, pass to use anti-aliasing. Anti-aliasing blurs your text slightly to make it look smoother. Figure 17-4 shows what a line (with enlarged pixels) looks like with and without anti-aliasing.

Attributes

The data type (called Rect for short) represent rectangular areas of a certain size and location. To create a new Rect object call the function . The parameters are integers for the XY coordinates of the top left corner, followed by the width and height, all in pixels.

The function name with the parameters looks like this:

Just like methods are functions that are associated with an object, **attributes** are variables that are associated with an object. The Rect data type has many attributes that describe the rectangle they represent. Table 17-2 is a list of attributes of a Rect object named .

The great thing about Rect objects is that if you modify any of these attributes, all the other attributes will automatically modify themselves also. For example, if you create a Rect object that is 20 pixels wide and 20 pixels high, and has the top left corner at the coordinates (30, 40), then the X-coordinate of the right side will automatically be set to (because 20 + 30 = 50).

However, if you change the attribute with the line , then Pygame will automatically change the attribute to 120 (because 20 + 100 = 120). Every other attribute for that Rect object is also updated.

The `get_rect()` Methods for `pygame.font.Font` and `pygame.Surface` Objects

Notice that both the Font object (stored in the variable on line 23) and the Surface object (stored in variable on line 24) both have a method called . Technically, these are two different methods. But the programmers of Pygame gave them the same name because they both do the same thing and return Rect objects that represent the size and position of the Font or Surface object.

The module you import is , and inside the module are the and modules. Inside those modules are the and data types. The Pygame programmers made the modules begin with a lowercase letter, and the data types begin with an uppercase letter. This makes it easier to distinguish the data types and the modules.

Constructor Functions

Create a object by calling a function named . The function has the same name as the data type. Functions that have the same name as their data type and create objects or values of this data type are called **constructor functions**.

The `fill()` Method for Surface Objects

You want to fill the entire surface stored in with the color white. The function will completely cover the entire surface with the color you pass as the parameter. (In this case, the variable is set to the value .

An important thing to know about Pygame is that the window you call the method or any of the other drawing functions. These will change the Surface
"
function is called.

This is becaus
modifying the image on the screen. It is much more efficient to draw onto the screen once after all of the drawing functions have drawn to the surface.

Pygame’s Drawing Functions

The `pygame.draw.polygon()` Function

A polygon is multisided shape with straight line sides. Circles and ellipses are not polygons. Figure 17-5 has some examples of polygons.

Table 17-2: Rect Attributes

`pygame.Rect` **Attribute**	**Description**
	Integer value of the X-coordinate of the left side of the rectangle.
	Integer value of the X-coordinate of the right side of the rectangle.
	Integer value of the Y-coordinate of the top side of the rectangle.
	Integer value of the Y-coordinate of the bottom side of the rectangle.
	Integer value of the X-coordinate of the center of the rectangle.
	Integer value of the Y-coordinate of the center of the rectangle.
	Integer value of the width of the rectangle.
	Integer value of the height of the rectangle.
	A tuple of two integers: (width, height)
	A tuple of two integers: (left, top)
	A tuple of two integers: (right, top)
	A tuple of two integers: (left, bottom)
	A tuple of two integers: (right, bottom)
	A tuple of two integers: (left, centery)
	A tuple of two integers: (right, centery)
	A tuple of two integers: (centerx, top)
	A tuple of two integers: (centerx, bottom)

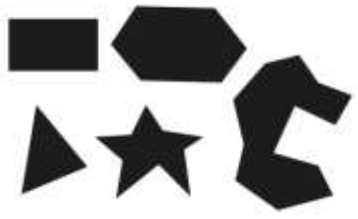


Figure 17-5: Examples of Polygons.

The function can draw any polygon shape you give it. The parameters, in order, are:

- The Surface object to draw the polygon on.
- The color of the polygon.
- A tuple of tuples that represents the XY coordinates of the points to draw in order. The last tuple will automatically connect to the first tuple to complete the shape.
- Optionally, an integer for the width of the polygon lines. Without this, the polygon will be filled in.

Line 31 draws a green pentagon on the Surface object.

The `pygame.draw.line()` Function

The parameters, in order, are:

- The Surface object to draw the line on.
- The color of the line.
- A tuple of two integers for the XY coordinate of one end of the line.
- A tuple of two integers for the XY coordinates of the other end of the line.
- Optionally, an integer for the width of the line.

If you pass for the width, the line will be four pixels thick. If you do not specify the parameter, it will take on the default value of . The three calls on lines 34, "

The `pygame.draw.circle()` Function

The parameters, in order, are:

- The Surface object to draw the circle on.
- The color of the circle.
- A tuple of two integers for the XY coordinate of the center of the circle.
- An integer for the radius (that is, the size) of the circle.
- Optionally, an integer for the width. A width of 0 means that the circle will be filled in.

Line 39 draws a blue circle on the Surface object.

The `pygame.draw.ellipse()` Function

The function is similar to the function. The parameters, in order, are:

- The Surface object to draw the ellipse on.
- The color of the ellipse.
- A tuple of four integers is passed for the left, top, width, and height of the ellipse.
- Optionally, an integer for the width. A width of 0 means that the circle will be filled in.

Line 42 draws a red ellipse on the Surface object.

The `pygame.draw.rect()` Function

The function will draw a rectangle. The third parameter is a tuple of four integers for the left, top, width, and height of the rectangle. Instead of a tuple of four integers for the third parameter, you can also pass a Rect object.

On line 45, you want the rectangle you draw to be 20 pixels around all the sides of the text. This is why you minus 20. (Remember, you subtract because coordinates decrease as you go left and up.) And the width and height are equal to the width and height of the plus 40 (because the left and top were moved back 20 pixels, so you need to make up for that space).

The `pygame.PixelArray` Data Type

Line 48 creates a object (called a PixelArray object for short). The PixelArray object is a list of lists of color tuples that represents the Surface object you passed it.

Line 48 passes to the call, so assigning to on line 49 will change the pixel at the coordinates (480, 380) to be a black pixel. Pygame will automatically modify the object with this change.

The first index in the PixelArray object is for the X-coordinate. The second index is for the Y-coordinate. PixelArray objects make it easy to set individual pixels on a PixelArray object to a specific color.

Creating a PixelArray object from a Surface object will lock that Surface object. Locked means that no function calls (described next) can be made on that Surface object. To unlock the Surface object, you must delete the PixelArray object with the operator. If you forget to delete the PixelArray "

.

The `blit()` Method for Surface Objects

The method will draw the contents of one Surface object onto another Surface object. and draws it to the Surface object stored in the variable.

The second parameter to specifies where on the windowSurface surface the text surface should be drawn. Pass the Rect object you got from calling on line 23.

The `pygame.display.update()` Function

In Pygame, nothing is actually drawn to the screen until the function is called. This is because drawing to the screen is slow compared to drawing on the Surface

een after each drawing function is called, but only update the screen once after all the drawing functions have been called.

Events and the Game Loop

In previous games, all of the programs print everything immediately until they reach a function call. At that point, the program stops and waits for the user to type something in and press ENTER. But Pygame programs are constantly running through a loop called the **game loop**. In this program, all the lines of code in the game loop execute about a hundred times a second.

The game loop is a loop that constantly checks for new events, updates the state of the window, and draws the window on the screen. **Events** are objects of the data type that are generated by Pygame whenever the user presses a key, clicks or moves the mouse, or makes some other event occur. (These events are listed on Table 18-1.)

Line 59 is the start of the game loop. The condition for the statement is set to so that it loops forever. The only time the loop exits is if an event causes the program to terminate.

The `pygame.event.get()` Function

Calling retrieves any new objects (called Event objects for short) that have been generated since the last call to . These events are returned as a list of Event objects. All Event objects have an attribute called which tell us what type of event it is. (In this chapter we only deal with the types of event. The other types of events are covered in the next chapter.)

Line 60 has a loop to iterate over each Event object in the list returned by . If the attribute of the event is equal to the constant variable , then you know the user has closed the window and wants to terminate the program.

Pygame generates the event (which was imported from the module) when the user clicks on the close button (usually an ×)
the computer is shutting down and tries to terminate all the running programs. For whatever reason the event was generated, you should terminate the program.

The pygame.quit() Function

If the event has been generated, the program should call both and
.

ve covered many new topics that we didn't have to deal with in our previous games. Even though the code is more
"
to create games with animated graphics that move.

Animation

In this program we have several different blocks bouncing off of the edges of the window. The blocks are different colors and sizes and move only in diagonal directions. To animate the blocks (that is, make them look like they are moving) we will move the blocks a few pixels over on each iteration through the game loop. This will make it look like the blocks are moving around the screen.

Source Code of the Animation Program

Type the following program into the file editor and save it as *animation.py*. If you get errors after typing this code in, compare the code you typed
http://invpy.com/diff/animation.

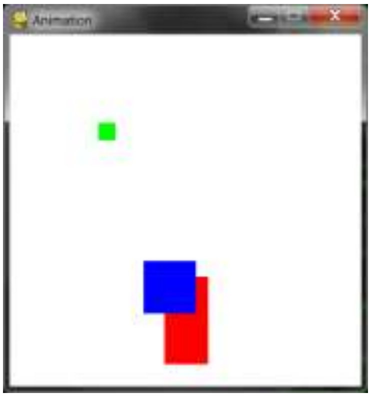


Figure 17-6: An altered screenshot of the Animation program.

How the Animation Program Works

In this program, we will have three different colored blocks moving around and bouncing off the walls. To do this, we need to first consider how we want the blocks to move.

Moving and Bouncing the Blocks

Each block will move in one of four diagonal directions. When the block hits the side of the window, it should bounce off the side and move in a new diagonal direction. The blocks will bounce as shown Figure 17-7.

The new direction that a block moves after it bounces depends on two things: which direction it was moving before the bounce and which wall it bounced off of. There are a total of eight possible ways a block can bounce: two different ways for each of the four walls.

For example, if a block is moving down and right, and then bounces off of the bottom edge of the
"

We can represent the blocks with a Rect object to represent the position and size of the block, a tuple of three integers to represent the color of the block, and an integer to represent which of the four diagonal directions the block is currently moving.

On each iteration in the game loop, adjust the X and Y position of the block in the Rect object. Also, in each iteration draw all the blocks on the screen at their current position. As the program execution iterates over the game loop, the blocks will gradually move across the screen so that it looks like they are smoothly moving and bouncing around on their own.

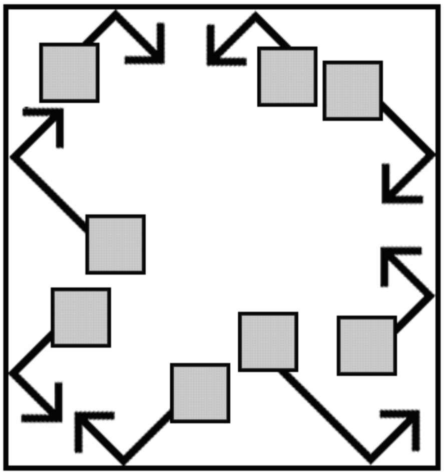


Figure 17-7: The diagram of how blocks will bounce.

Creating and Setting Up Pygame and the Main Window

"

just the call to . Use constant variables so that if you ever want to change the size of the window, you only have to change lines 8 and 9. Since the window width and height never

"

by calling .

Setting Up Constant Variables for Direction

We will use the keys on the number pad of the keyboard to remind us which belongs to which direction. This is similar to the Tic Tac Toe game. is down and left, is down and right, is up and left, and is up and right. However, it may be hard to remember this, so instead use constant variables instead of these integer values.

You could have used any value you wanted for these directions instead of using a constant variable. For example, you could use the string to represent the down and left diagonal direction. However, if you ever mistype the string (for example, as), Python would not recognize that you meant to type instead of . This bug would cause your program to behave strangely, but the program would not crash.

But if you use constant variables, and accidentally type the variable name instead of the name , Python would notice that no such variable named and crash the program with an error. This would still be a pretty bad bug, but at least you would know about it immediately and could fix it.

Use a constant variable to determine how fast the blocks should move. A value of here means that each block will move 4 pixels on each iteration through the game loop.

Setting Up Constant Variables for Color

Lines 22 to 25 set up constant variables for the colors. Remember, Pygame uses a tuple of three integer values for the amounts of red, green, and blue called an RGB value. The integers are from to .

T you use a variable named for the color green. It is easier to know that stands for the color green, rather than .

Setting Up The Block Data Structures

Set up a dictionary as a data structure that represents each block. (Chapter 9½ introduced dictionaries.) The dictionary will have the keys of (with a Rect object for a value), (with a tuple of three integers for a value), and (with one of the direction constant variables for a value).

The variable will store one of these block data structures. This block has its top left corner located at an X-coordinate of 300 and Y-coordinate of 80. It has a width of 50 pixels and a height of 100 pixels. Its color is red and its direction is set to .

Line 29 and 30 creates two more similar data structures for blocks that are different sizes, positions, colors, and directions.

Line 31 put all of these data structures in a list, and store the list in a variable named .

The variable stores a list. would be the dictionary data structure in . would be the key in , so the expression would evaluate to . This way you can refer to any of the values in any of the block data structures by starting with .

Running the Game Loop

Inside the game loop, the blocks will move around the screen in the direction that they are going and bounce if they have hit a side. There is also code to draw all of the blocks to the surface and call .

The loop to check all of the events in the list returned by is the same as .

First, line 42 fills the entire surface with black so that anything previously drawn on the surface is erased.

Moving Each Block

Next, the code must update the position of each block, so iterate over the list. Inside the loop, ll refer to the current block as simply so it will be easy to type.

The new value to set the and attributes to depends on the direction. If the direction of the block (which is stored in the key) is either or , you want to *increase* the attribute. If the direction is or , you want to *decrease* the attribute.

If the direction of the block is or , you want to *increase* the attribute. If the direction is or , you want to *decrease* the attribute.

Change the value of these attributes by the integer stored in . stores how many pixels over blocks move on each iteration of the game loop, and was set on line19.

Checking if the Block has Bounced

After lines 44 to 57 move the block, check if the block has gone past the edge of the window. If it
"

key. The block will move in the new direction on the next iteration of the game loop. This makes it look like the block has bounced off the side of the window.

statement"
attribute is less than . In that case, change the direction based on what direction the block was moving (either or).

Changing the Direction of the Bouncing Block

Look at the bouncing diagram earlier in this chapter. To move past the top edge of the window, the block had to either be moving in the or directions. If the block was moving in the direction, the new direction (according to the bounce diagram) will be . If the block was moving in the direction, the new direction will be .

Lines 66 to 71 handles if the block has moved past the bottom edge of the window. They check if the attribute (not the attribute) is *greater* than the value in . Remember that the Y-coordinates start at at the top of the window and increase to at the bottom.

The rest of the code changes the direction based on what the bounce diagram in Figure 17-7 says.

Lines 78 to 83 are similar to lines 72 to 77, but checks if the left side of the block has moved to the left of the left edge of the window. Remember, the X-coordinates start at on the left edge of the window and increase to on the right edge of the window.

Drawing the Blocks on the Window in Their New Positions

Now that the blocks have moved, they should be drawn in their new positions on the surface by calling the function. Pass because it is the Surface object to draw the rectangle on. Pass the because it is the color of the rectangle. Pass because it is the Rect object with the position and size of the rectangle to draw.

Line 86 is the last line of the loop. If you wanted to add new blocks, you only have to modify the list on line 31 and the rest of the code still works.

Drawing the Window on the Screen

After each of the blocks in the list has been drawn, call so that the surface is draw on the screen.

After this line, the execution loops back to the start of the game loop and begin the process all over again. This way, the blocks are constantly moving a little, bouncing off the walls, and being drawn on the screen in their new positions.

The call to the function is there because the computer can move, bounce, and draw the blocks so fast that if the program ran at full speed, all the blocks would look like a blur. (Try commenting out the line and running the program to see this.)

This call to will stop the program for 0.02 seconds, or 20 milliseconds.

Drawing Trails of Blocks

Comment out line 42 (the line) by adding a to the front of the line. Now run the program.

Without the call to , you don t black out the entire window before drawing the rectangles in their new position. The trails of rectangles appear because the old

Remember that the blocks are not really moving. On each iteration through the game loop, the code redraws the entire window with new blocks that are located a few pixels over each time.

Summary

This chapter has presented a whole new way of creating computer programs. The previous
" n our animation program, the program is constantly updating the data structures of things without waiting for input from the player.

Remember in our Hangman and Tic Tac Toe games we had data structures that would represent the state of the board, and these data structures would be passed to a function to be displayed on the screen. Our animation program is similar. The variable holds a list of data structures representing blocks to be drawn to the screen, and these are drawn to the screen inside the game loop.

But without calls to , how do we get input from the player? In our next chapter, we will cover how programs can know when the player presses keys on the keyboard. We will also learn of a concept called collision detection.

Topics Covered In This Chapter:

- Collision Detection
-
- Keyboard Input in Pygame
- Mouse Input in Pygame

Collision detection is figuring when two things on the screen have touched (that is, collided with) each other. For example, if the player touches an enemy they may lose health. Or the program needs to know when the player touches a coin so that they automatically pick it up. Collision detection can help determine if the game character is standing on solid ground or if nothing but empty air underneath them.

In our games, collision detection will determine if two rectangles are overlapping each other or not. Our next example program will cover this basic technique.

"

n calling the
function like we did for our text programs. But using the keyboard is much more interactive in

will make your games more exciting!

Source Code of the Collision Detection Program

Much of this code is similar to the animation program, so the explanation of the moving and bouncing code is skipped. (See the animation program in Chapter 17 for that.) A bouncer will bounce around the window. A list of Rect objects will represent food squares.

On each iteration through the game loop, the program will read each Rect object in the list and draw a green square on the window. Every forty iterations through the game loop we will add a new Rect object to the list so that the screen constantly has new food squares in it.

The bouncer is represented by a dictionary. The dictionary has a key named (whose value is a object) and a key named (whose value is one of the constant direction varia

As the bouncer bounces around the window, we check if it collides with any of the food squares. If it does, we delete that food square so that it will no longer be drawn on the screen. This will make i

Type the following into a new file and save it as *collisionDetection.py*. If you get errors after typing this code in, compare the code you typed
http://invpy.com/diff/collisionDetection.

The program will look like Figure 18-1. The the bouncer square will bounce around the window. When it collides with the green food squares they will disappear from the screen.

Figure 18-1: An altered screenshot of the Collision Detection program.

Importing the Modules

The collision detection program imports the same things as the Animation program in the last chapter, along with the module.

The Collision Detection Algorithm

To do collision detection, you need a function that can determine if two rectangles collide with each other or not. Figure 18-2 shows colliding and non-colliding rectangles.

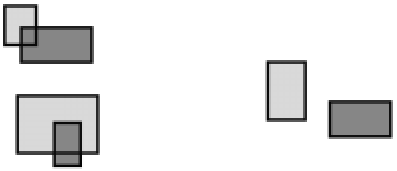


Figure 18-2: Examples of colliding rectangles (left) and rectangles that don’t collide (right).

is passed two objects. The function will return if they do and There is a simple rule to follow to determine if rectangles collide. Look at each of the four corners on both rectangles. If at least one of these eight corners is inside the other rectangle, then you know that the two rectangles have collided. You can use this fact to determine if returns or .

inside another. Later you will create a function called that returns if the XY coordinates of the point are inside the rectangle. Call this function for each of the eight corners, and if any of these calls return , the operators will make the entire condition .

The parameters for are and . First check if inside , then check if .

You d need to repeat the code that checks all four corners for both and . Instead, use and on lines 7 to 10. The loop on line 5 uses multiple assignment. On the first iteration, is set to and is set to . On the second iteration through the loop, it is the opposite: is set to and is set to .

Line 11 never returns , then none of the eight corners checked are in the other rectangle. In " line 13 returns .

Determining if a Point is Inside a Rectangle

The function is called from . The function will return if the XY coordinates passed are located inside the object passed as the third parameter. Otherwise, this function returns .

Figure 18-3 is an example picture of a rectangle and several points. The points and the corners of the rectangle are labeled with coordinates.

A point is inside the rectangle if the following four things are true:

- The X-coordinate is greater than the X-
- The X-coordinate is less than the X-coo
- The Y-coordinate is greater than the Y-
- The Y-coordinate is less than the Y-

If any of those parts are , then the point is outside the rectangle. Line 16 combines all four of these conditions into the operators.

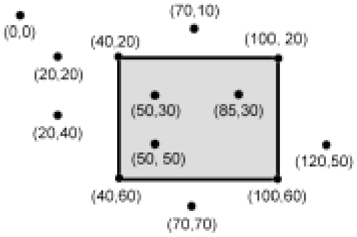


Figure 18-3: Example of coordinates inside and outside of a rectangle. The (50, 30), (85, 30) and (50, 50) points are inside the rectangle, and all the others are outside.

This function is called from the function to see if any of the corners in the two objects are inside each other. These two functions give you the power to do collision detection between two rectangles.

The pygame.time.Clock Object and tick() Method

Much of lines 22 to 43 do the same things that the Animation program in the last chapter did: initialize Pygame, set and , and assign the color and direction constants.

However, line 24 is new:

In the previous Animation program, a call to would slow down the program lem with is that might be too much of a pause on slow computers and not enough of a pause on fast computers.

A object can pause an appropriate amount of time on any computer. Line 125 calls inside the game l method waits enough time so that it runs at about 40 iterations a second, no matter what the A call to should only appear once in the game loop.

Setting Up the Window and Data Structures

Lines 46 to 48 set up a few variables for the food blocks that appear on the screen. will start at the value , at , and at .

Line 49 sets up a new data structure called . is a dictionary with two keys. The key has a

The key has a direction that the bouncer is currently moving. The bouncer will move the

The program will keep track of every food square with a list of Rect objects in . Lines 51 and 52 create twenty food squares randomly placed around the screen. You can use the function to come up with random XY coordinates.

On line 52, we will call the constructor function to return a new object. It will represent the position and size of the food square. The first two parameters for are the XY coordinates of the top left corner. You want the random coordinate to be between 0 and the size of the window minus the size of the food square. If you had the random coordinate between 0 and the size of the window, then the food square might be pushed outside of the window altogether, like in Figure 18-4.

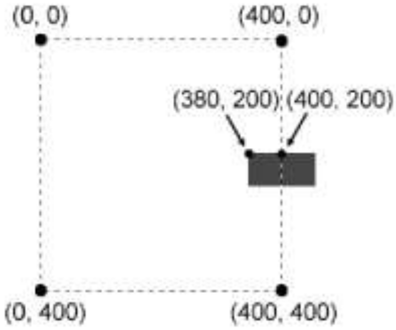


Figure 18-4: For a 20 by 20 rectangle, having the top left corner at (400, 200) in a 400 by 400 window would place the rectangle outside of the window. To be inside, the top left corner should be at (380, 200) instead.

The third parameter for is a tuple that contains the width and height of the food square. Both the width and height is the value in the constant.

Drawing the Bouncer on the Screen

Lines 71 to 109 cause the bouncer to move around the window and bounce off of the edges of the window. This code is similar to lines 44 to 83 of the Animation program in the last chapter, so the explanation will be skipped.

After moving the bouncer, line 112 draws it in its new position. The passed for the first parameter tells Python which Surface object to draw the rectangle on. The variable, which has stored in it, will tell Python to draw a white rectangle. The Rect object stored in the dictionary at the key tells the position and size of the rectangle to draw.

Colliding with the Food Squares

Before drawing the food squares, check if the bouncer has overlapped any of the food squares. If it has, remove that food square from the list. This way, Python draw any food

On each iteration through the loop, the current food square from the (plural) list is in the variable (singular).

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

Notice that a slight difference with this loop. If you look carefully at line 116, it i iterating over but actually over .

Remember how slices work. evaluates to a copy of the list with the items from the start and up to (but not including) the item at index 2. evaluates to a copy of the list with the items from index 3 to the end of the list.

will give you a copy of the list with the items from the start to the end. Basically, creates a new list with a copy of all the items in . This is a shorter way to copy a list than, say, what the function does in the previous Tic Tac Toe game.

You cannot add or remove items from a list while you are iterating over it. Python can lose track of what the next value of variable should be if the size of the list is always changing. Think of how difficult it would be to count the number of jelly beans in a jar while someone was adding or removing jelly beans.

But if you iterate over a copy of the list (and the copy never changes), adding or removing items from the original list be a problem.

Removing the Food Squares

Line 116 is where comes in handy. If the bouncer and the current food square two rectangles overlap, then will return and line 117 removes the overlapping food square from the list.

Drawing the Food Squares on the Screen

The code on lines 120 and 121 are similar to how we drew the white square for the player. Line 120 loops through each food square in the list. Line 121 draws the food square onto the surface. This program was similar to the bouncing program in the previous
"

This program is almost identical to the collision detection program. But in this program, the bouncer only moves around when the user holds down the arrow keys on the keyboard.

You can also click anywhere in the window and create new food objects. In addition, the ESC key player to a random place on the screen.

Setting Up the Window and Data Structures

Starting at line 29, the code sets up some variables that track the movement of the bouncer.

The four variables have Boolean values to keep track of which of the arrow keys are being held down. For example, when the user pushes the left arrow key on their keyboard, is set to . When they let go of the key, is set back to .

Lines 34 to 43 are identical to code in the previous Pygame programs. These lines handle the start

this code here since we have already covered it in the last chapter.

Events and Handling the *KEYDOWN* Event

The code to handle the key press and key release events start on line 44. At the start of the program, they are all set to .

Pygame has an event type called . This is one of the other events that Pygame can generate. A brief list of the events that could be returned by is in Table 18-1.

Table 18-1: Events and when they are generated.

Event Type	Description
	Generated when the user closes the window.
	Generated when the user presses down a key. Has a attribute that tells which key was pressed. Also has a attribute that tells if the Shift, Ctrl, Alt, or other keys were held down when this key was pressed.
	Generated when the user releases a key. Has a and attribute that are similar to those for .
	Generated whenever the mouse moves over the window. Has a attribute that returns tuple (x, y) for the coordinates of where the mouse is in the window. The attribute also returns a (x, y) tuple, but it gives coordinates relative since the last event. For example, if the mouse moves left by four pixels from (200, 200) to (196, 200), then will be the tuple value . The 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 a middle mouse button), and the third integer is for the right mouse button. These integers will be if they are not being pressed down when the mouse moved and if they are pressed down.
	Generated when a mouse button is pressed down in the window. This event has a attribute which is an (x, y) tuple for the coordinates of where the mouse was when the button was pressed. There is also a attribute which is an integer from to that tells which mouse button was pressed, explained in Table 18-2.
	Generated when the mouse button is released. This has the same attributes as .

Table 18-2: The attribute values and mouse button.

Value of button	Mouse Button
1	Left button
2	Middle button
3	Right button
4	Scroll wheel moved up
5	Scroll wheel moved down

Setting the Four Keyboard Variables

If the event type is , then the event object will have a attribute that tells which key was pressed down. Line 46 compares this attribute to , which is the constant that represents the left arrow key on the keyboard. Lines 46 to 57 do similar checks for each of the other arrow keys: , , , .

When one of these keys is pressed down, set the corresponding movement variable to . Also, set the movement variable of the opposite direction to .

For example, the program executes lines 47 and 48 when the left arrow key has been pressed. In this case, set to and to (even though might already be , set it to just to be sure).

On line 46, in can either be equal to or . The value in is set to the integer ordinal value of the key that was pressed on the keyboard. (There is no ordinal value for the arrow keys, which is why we use the constant variable .) You can use the function to get the ordinal value of any single character to compare it with .

By executing the code on lines 47 and 48 if the keystroke was either or , you make the left arrow key and the A key do the same thing. The W, A, S, and D keys are all used as alter -
you use your left hand. The arrow keys can be pressed with your right hand.

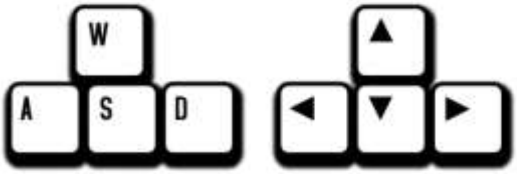


Figure 18-5: The WASD keys can be programmed to do the same thing as the arrow keys.

Handling the KEYUP Event

When the user releases the key that they are holding down, a event is generated.

If the key that the user released was the ESC key, then terminate the program. Remember, in Pygame you must call the function before calling the function.

Lines 62 to 69 will set a movement variable to let go.

Teleporting the Player

"

Handling the MOUSEBUTTONUP Event

Mouse input is handled by events just like keyboard input is. The event occurs when the user releases the mouse button after clicking it. The attribute in the Event object is set to a tuple of two integers for the XY coordinates for where the mouse cursor was at the time of the click.

On line 75, the X-coordinate is stored in and the Y-coordinate is stored in . Line 75 creates a new Rect object to represent a new food and place it where the event occurred. By adding a new Rect object to the list, the code will display a new food square is displayed on the screen.

Moving the Player Around the Screen

, , , and) to or

represented by the object stored in) by adjusting XY coordinates of .

If is set to
the window), then line 88 moves
current attribute. Lines 89 to 94 do the same thing for the other three directions.

The `colliderect()` Method

In the previous Collision Detection program, the function to check if one rectangle had collided with another. That function was included in this book so you could understand how the code behind collision detection works.

In this program, you can use the collision detection function that comes with Pygame. The method for objects is passed another object as an argument and returns if the two rectangles collide and if they do not.

The rest of the code is similar to the code in the Input and Collision Detection programs.

Summary

This chapter introduced the concept of collision detection, which is in many graphical games. Detecting collisions between two rectangles is easy: check if the four corners of either rectangle are within the other rectangle. This is such a common thing to check for that Pygame provides its own collision detection method named for objects.

The first several games in this book were text-based. The program output was text printed to the screen and the input was text typed by the user on the keyboard. But graphical programs can accept keyboard and mouse inputs.

Furthermore, these programs can respond to single keystrokes when the user pushes down or lets
ENTER. This allows
for immediate feedback and much more interactive games.

Topics Covered In This Chapter:

- Sound and Image Files
- Drawing Sprites
- The Function
- The Data Type
- The Module

"

different shapes. In this chapter, we will learn how to show pictures and images (called sprites) and play sounds and music in our games.

A **sprite** is a name for a single two-dimensional image that is used as part of the graphics on the screen. Figure 19-1 shows some example sprites.

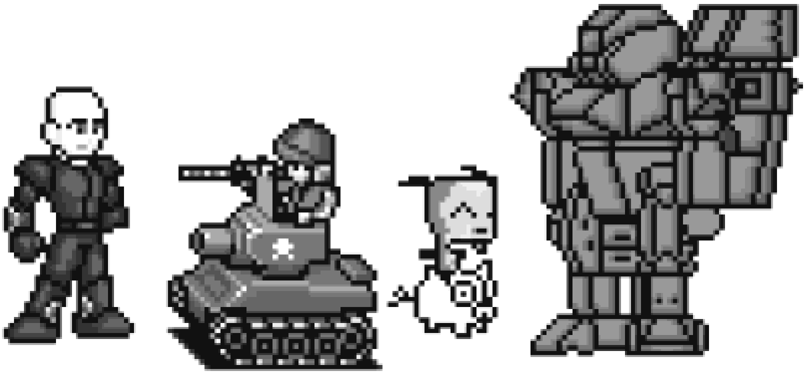


Figure 19-1: Some examples of sprites.

Figure 19-2 shows being used in a complete scene.

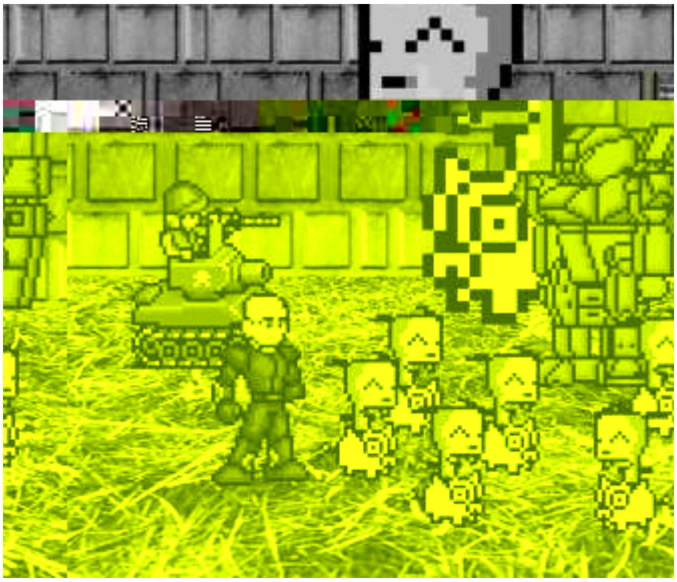


Figure 19-2: An example of a complete scene, with sprites drawn on top of a background.

The sprite images are drawn on top of a background. Notice that you can flip the sprite image horizontally so that the sprites are facing the other way. You can draw the same sprite image multiple times on the same window. You can also resize the sprites to be larger or smaller than the original sprite image. The background image can be considered one large sprite.

The next program will demonstrate how to play sounds and draw sprites using Pygame.

Sound and Image Files

Sprites are stored in image files on your computer. There are several different image formats that Pygame can use. You can tell what format an image file uses by looking at the end of the file name (after the last period). This is called the **file extension**. For example, the file *player.png* is in the PNG format. The image formats Pygame supports include BMP, PNG, JPG, and GIF.

You can download images from your web browser. On most web browsers, you have to right-click on the image in the web page and select Save from the menu that appears. Remember where on the hard drive you saved the image file. Copy this downloaded image file into the same folder *.py* file. You can also create your own images with a drawing program like MS Paint or Tux Paint.

The sound file formats that Pygame supports are MID, WAV, and MP3. You can download sound effects from the Internet just like image files. They must be in one of these three formats. If your computer has a microphone, you can also record sounds and make your own WAV files to use in your games.

Sprites and Sounds Program

This program is the same as the Keyboard and Mouse Input program from the last chapter. However, in this program we will use sprites instead of plain looking squares. We will use a sprite of a little person instead of the white player square, and a sprite of cherries instead of the green food squares. We also play background music and a sound effect when the player sprite eats one of the cherry sprites.

Source Code of the Sprites and Sounds Program

If you know how to use graphics software such as Photoshop or MS Paint, you can draw your

"

websites and use those image files instead. The same applies for music and sound files. You can also find images on websites or images from a digital camera. You can download the image and

http://invpy.com/downloads.

If you get errors after typing this code in, compare the code you typed
online diff tool at http://invpy.com/diff/spritesAndSounds.

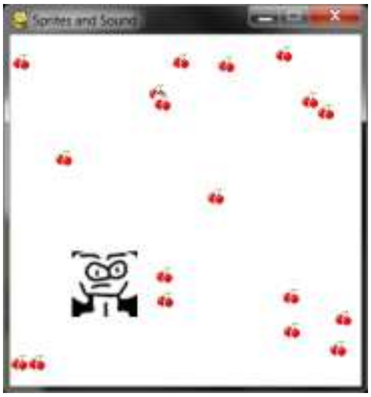


Figure 19-3: An altered screenshot of the Sprites and Sounds game.

Setting Up the Window and the Data Structure

Most of the code in this program is the same as the Collision Detection program in the previous

" scribes this program on line 12. Pass the string to the function.

We are going to use three different variables to represent the player, unlike the previous programs that just used one.

The variable on line 18 will store a Rect object that keeps track of where and how big the player is. The " location. At the beginning of the program, the top left corner of the player is located at (300, 100) and the player will have a height and width of 40 pixels to start.

The second variable on line 19 that represents the player is . The function is passed a string of the filename of the image to load. The return value is a Surface object that has the graphics in the image file drawn on its surface. We store this Surface object inside of .

The third variable is explained in the next section.

The `pygame.transform.scale()` Function

On line 20, we will use a new function in the module. The function can shrink or enlarge a sprite. The first argument is a 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 function returns a object with the image drawn at a new size. We will store the original image in the variable but the stretched image in the variable.

On line 21, we call again to create a Surface object with the cherry image drawn on it. Be sure you have the *player.png* and *cherry.png* files in the same directory as the *spritesAndSounds.py* file, otherwise Pygame be able to find them and will give an error.

Setting Up the Music and Sounds

Next you need to load the sound files. There are two modules for sound in Pygame. The module can play short sound effects during the game. The module can play background music.

Call the constructor function to create a object (called a Sound object for short). This object has a method that when called will play the sound effect when called.

Line 39 calls to load the background music. Line 40 calls to start playing the background music. The first parameter tells Pygame how many times to play the background music after the first time we play it. So passing will cause Pygame to play the background music 6 times. is a special value, and passing it for the first parameter makes the background music repeat forever.

The second parameter to is the point in the sound file to start playing. Passing will play the background music starting from the beginning. Passing for the second parameter will start the background music two and half seconds from the beginning.

Finally, the variable will have a Boolean value that tells the program if it should

the program without the sound playing.

Toggling the Sound On and Off

The M key will turn the background music on or off. If is set to , then the background music is currently playing and we should stop the music by calling . If is set to , then the backgr currently playing and should be started by calling .

Finally, no matter what, we want to toggle the value in . **Toggling** a Boolean value means to set to the opposite of its current value. The line sets the variable to if it is currently or sets it to if it is currently . Think of toggling as what happens when you flip a light switch on or off: toggling the light switch sets it to the opposite setting.

Drawing the Player on the Window

Remember that the value stored in is a Surface object. Line 110 draws the sprite of the player onto th).

The second parameter to the method is a Rect object that specifies where on the Surface object the sprite should be blitted. The Rect object stored in is what keeps track of the position of the player in the window.

Checking if the Player Has Collided with Cherries

This code is similar to the code in the previous programs. But there are a couple of new lines. Call the method on the Sound object stored in the variable. But only do this if is set to (which means that the sound is turned on).

When the player eats one of the cherries, the size of the player increases by two pixels in height and width. On line 116, a new Rect object that is 2 pixels larger than the old Rect object will be the new value of .

While the Rect object represents the position and size of the player, the image of the player is stored in a as a Surface object. Create a new stretched image by calling . Be sure to pass the original Surface object in and not .

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

Topics Covered In This Chapter:

- The flag
- Pygame constant variables for keyboard keys
- The Rect method
- The function
- Implementing cheat codes
- Modifying the Dodger game

The last three chapters went over the Pygame module and demonstrated how to use its many
" d Dodger.

who must dodge a whole bunch of baddies that fall from the top of the screen. The longer the player can keep dodging the baddies, the higher the score they will get.

" "
"
baddies will reverse their direction and travel up the screen instead of downwards.

Review of the Basic Pygame Data Types

- -
  location can be determined by the Rect objec attribute (or the ,
  , and attributes). These corner attributes are a tuple of integers for the X- and Y-coordinates. The size can be determined by the and attributes, which are integers of how many pixels long or high the rectangle area is. Rect objects have a method to check if they are colliding with another Rect object.

- - 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 method that is used to draw the image on one Surface object onto another Surface object. The Surface object returned by the function is special because anything drawn on that is called.

- - The module generates Event objects whenever the user provides keyboard, mouse, or another kind of input. The function returns a list of these Event objects. You can check what type of event the Event object is by checking its attribute. , , and are examples of some event types.
- - The module has the Font data type which represents the typeface used for text in Pygame. The arguments to pass to are a string of the font name and an integer of the font size. However it is common to pass for the font name to get the default system font.
- - The Clock object in the module is helpful for keeping our games from running as fast as possible. The Clock object has a method, which we pass how many frames per second (FPS) we want the game to run at. The higher the FPS, the faster the game runs.

Type in the following code and save it to a file named *dodger.py*. This game also requires some other image and sound files, which you can download from the URL http://invpy.com/downloads.

Source Code of Dodger

You can download this code from the URL http://invpy.com/chap20. If you get errors after typing this code in, compare the code you typed http://invpy.com/diff/dodger.

When you run this program, the game will look like Figure 20-1.

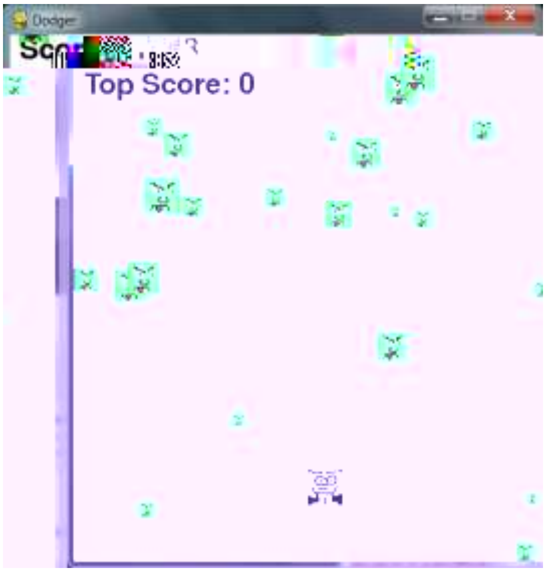


Figure 20-1: An altered screenshot of the Dodger game.

Importing the Modules

The Dodger game imports the same modules previous Pygame programs have: , , , and . The module contains several constant variables that Pygame uses such as the event types (, , etc.) and keyboard keys (, , etc.). By using the syntax, you can just type in the source code instead of .

Setting Up the Constant Variables

The constant variables on lines 4 to 14 are much more descriptive than typing out the values. For example, from the line is more understandable than .

You can easily change the game by changing the constant variables. By changing on line 4, you automatically change the code everywhere is used. If you had used the value instead, then you would have to change each occurrence of in the code. It is easier to change the value in the constant once.

The method call on line 189 will slow the game down enough to be playable. You pass an integer to so that the function knows how long to pause the program. This integer (which you store in) is the number of frames per second you want the game to run.

You can set to , and always call . Then you can change to a higher value to have the game run faster or a lower value to slow the game down.

Lines 9 to 13 set some more constant variables that will describe the falling baddies. The width and height of the baddies will be between and . The rate at which the baddies fall down the screen will be between and pixels per iteration through the game loop. And a new baddie will be added to the top of the window every iterations through the game loop.

The
on each iteration through the game loop if the character is moving. By increasing this number, you can increase the speed the character moves.

Defining Functions

Pygame requires that you call both and . Put them both into a function called . Now you only need to call , instead of both of the and functions.

l the player presses a key. Create a new function called . Inside this function, an infinite loop that only breaks when a or event is received. At the start of the loop, to return a list of Event objects to check out.

If the player has closed the window while the program is waiting for the player to press a key, Pygame will generate a event. In that case, call the function on line 24.

If you receive a event, then you should first check if it is the ESC key that was pressed. If the player presses the ESC key, the program t the case, then execution will skip the if-block on line 27 and go straight to the statement, which exits the function.

If a or " Since the loop does nothing, this will make it look like the game has frozen until the player presses a key.

The function will return one of the baddies. The dictionary data structures. Each of these dictionaries has a key, and the value for that key is a Rect object that represents the

is also a Rect object. Rect objects have a method named that returns if the Rect object has collided with the Rect object that is passed to it. Otherwise, will return .

The loop on line 31 iterates through each baddie dictionary in the list. If any of " will return . If the code manages to iterate through all the baddies in the list without detecting a collision with any of them, it will return .

Line 43 sets up the Pygame by calling the function. Line 44 creates a object and stores it in the variable. This object will help us keep the program from running too fast.

Line 45 creates a new Surface object which is used for the window displayed on the screen. You can specify the width and height of this Surface object (and the window) by passing a tuple with the and constant variables. Notice that only one argument passed to : a tuple. The arguments for are not two integers but one tuple of two integers.

Line 46 sets the caption of the window to the string . This caption will appear in the title bar at the top of the window.

"

" e way of

Calling will tell Pygame to make the cursor not visible.

Fullscreen Mode

The function has a second, optional parameter. You can pass the constant to make the window take up the entire screen instead of being in a window. Look at this modification to line 45:

It will still be and in size for the windows width and height, but the image will be stretched larger to fit the screen. Try running the program wiuth and without fullscreen mode.

Line 49 creates a Font object to use by calling . Passing uses the default font. Passing makes the font have a size of 48 points.

Next, create the Sound objects and set up the background music. The background music will constantly be playing during the game, but Sound objects will only be played when the player loses the game.

You can use any *.wav* or *.mid* file for this game. Some sound files website at http://invpy.com/downloads. Or you can use your own sound files for this game, as long as they have the filenames of *gameover.wav* and *background.mid*. (You can change the strings used on lines 53 and 54 to match the filenames.)

The constructor function creates a new Sound object and stores a reference to this object in the variable. In your own games, you can create as many Sound objects as you like, each with a different sound file.

The function loads a sound file to play for the background music. return any objects, and only one background sound file can be loaded at a time.

to be screen. The image for the character is stored in *player.png* and the image for the baddies is stored in *baddie.png*. All the baddies look the same, so you only need one image file for them. You can download these images from this http://invpy.com/downloads.

Display the Start Screen

When the game first starts, display the name on the screen. You also want to instruct the player that they can start the game by pushing any key. This screen appears so that the player has time to get ready to start playing after running the program.

On lines 62 and 63, call the function and pass it five arguments:

1) The string of the text you want to appear.

2) The font that you want the string to appear in.

3) The Surface object onto which to render the text.

4) The X coordinate on the Surface object to draw the text at.

5) The Y coordinate on the Surface object to draw the text at.

This may seem like many arguments to pass for a function call, but keep in mind that this function call replaces five lines of code each time you call it. This shortens the program and makes it easier to find bugs since less code to check.

The function will pause the game by looping until a event is generated. Then the execution breaks out of the loop and the program continues to run.

Start of the Main Game Code

The value in the variable starts at when the program first runs. Whenever the player loses and has a score larger than the current top score, the top score is replaced with this larger score.

The infinite loop started on line 69 is te The game loop handles events and drawing the window while the game is running. Instead, this loop will iterate
"
execution will loop back to line 69.

At the beginning, you want to set to an empty list. The variable is a list of dictionary objects with the following keys:

- - The Rect object that describes where and what size the baddie is.
- - How fast the baddie falls down the screen. This integer represents pixels per iteration through the game loop.
- - The Surface object that has the scaled baddie image drawn on it. This is the Surface object that is blitted to the Surface object returned by .

.

The starting location of the player is in the center of the screen and 50 pixels up from the bottom. -coordinate of the left edge. The second item in the tuple is the Y-coordinate of the top edge.

The movement variables , , , and are set to . The and variables are also set to . They will be set to only when "

The variable is a counter to tell the program when to add a new baddie at the top of the screen. The value in increments by one each time the game loop iterates.

When is equal to , then the counter resets to 0 and a new baddie is added to the top of the screen. (This check is done later on line 130.)

The background music starts playing on line 77 with a call to . The first argument is the number of times the music should repeat itself. is a special value that tells Pygame you want the music to repeat endlessly.

The second argument is a float that says how many seconds into the music you want it to start playing. Passing means the music starts playing from the beginning.

The Game Loop

ing the position of the player and baddies, handling events generated by Pygame, and drawing the game world on "

of the game loop. The longer the player can go without losing, the higher their score. The loop will only exit when the player either loses the game or quits the program.

Event Handling

There are four different types of events the program will handle: , , , and .

Line 82 is the start of the event-handling code. It calls , which returns a list of Event objects. Each Event object represents an event that has happened since the last call to . The code will check the attribute of the event object to see what type of event it is, and handle the event accordingly.

If the attribute of the Event object is equal to , then the user has closed the program. The constant variable was imported from the module.

, the player has pressed down a key. The Event object for keyboard events will also have a attribute that is set to the integer ordinal value of the key pressed. The function will return the ordinal value of the letter passed to it.

"

. If this condition is , set the variable to to

indicate that the reverse cheat
to activate the slow cheat.

ordinal values of lowercase letters, not uppercase.
Always use instead of . Otherwise, your

Lines 91 to 102 check if the event was generated by the player pressing one of the arrow or WASD ordinal value for every key on the keyboard, such as the arrow keys or the ESC key. Instead, the module provides constant variables to use instead.

Line 91 checks if the player has pressed the left arrow key with . Notice that pressing down on one of the arrow keys not only sets a movement variable to , but it also sets the movement variable in the opposite direction to .

For example, if the left arrow key is pushed down, then the code on line 93 sets to , but it also sets to . This prevents the player from confusing the program ections at the same time.

Table 20-1 lists commonly-used constant variables for the attribute of keyboard-related Event objects.

Table 20-1: Constant Variables for Keyboard Keys

Pygame Constant Variable	Keyboard Key	Pygame Constant Variable	Keyboard Key
	Left arrow		Home
	Right arrow		End
	Up arrow		PgUp

	Down arrow		PgDn
	Esc		F1
	Backspace		F2
	Tab		F3
	Return or Enter		F4
	Space bar		F5
	Del		F6
	Left Shift		F7
	Right Shift		F8
	Left Ctrl		F9
	Right Ctrl		F10
	Left Alt		F11
	Right Alt		F12

The event is created whenever the player stops pressing down on a keyboard key and releases it. Event objects with a type of also have a attribute just like events.

Line 105 checks if the player ha "
that case, line 106 sets to and line 107 resets the score to . The score reset is to discourage the player for using the cheats.

Lines 108 to 110 do the same thing for
released, is set to .

At any time during the game, the player can press the ESC key on the keyboard to quit. Line 14 checks if the key that was released was the ESC key by checking . If so, line 112 calls the function to exit the program.

Lines 114 to 121 check if the player has stopped holding down one of the arrow or WASD keys. In that case, the code sets the corresponding movement variable to .

For example, if the player was holding down the left arrow key, then the would have been set to on line 93. When they release it, the condition on line 114 will evaluate to , and the variable will be set to .

The move_ip() Method

"

button, but it does respond when the player moves the mouse. This gives the player two ways of controlling the player character in the game: the keyboard or the mouse.

The event is generated whenever the mouse is moved. Event objects with a set to also have an attribute named for the position of the mouse event. The attribute stores a tuple of the X- and Y-coordinates of where the mouse cursor moved in the " position of the mouse cursor.

The method for Rect objects will move the location of the Rect object horizontally or vertically by a number of pixels. For example, would move the Rect object 10 pixels to the right and 20 pixels down. To move the Rect object left or up, pass negative values. For example, will move the Rect object left by 5 pixels and up 15 pixels.

stands for

Rect object itself, rather than return a new Rect object with the changes. There is also a

"

the new location.

Adding New Baddies

On each iteration of the game loop, increment the variable by one. This only happens if the cheats are not enabled. Remember that and are set to

"

And while those keys are being held down, ncremented. Therefore, no new baddies will appear at the top of the screen.

When the reaches the value in , it is time to add a new baddie to the top of the screen. First, the counter is reset back to .

Line 132 generates a size for the baddie in pixels. The size will be a random integer between and , which are constants set to and on lines 9 and 10.

Line 133 is where a new baddie data structure is created. Remember, the data structure for is simply a dictionary with keys , , and . The key holds a reference to a Rect object which stores the location and size of the baddie. The call to the constructor function has four parameters: the X-coordinate of the top edge of the area, the Y-coordinate of the left edge of the area, the width in pixels, and the height in pixels.

The baddie needs to appear randomly across the top of the window, so pass for the X-coordinate of the left edge. The reason you pass instead of is because this value is for the left edge of the baddie. If the left edge of the baddie is too far on the right side of the screen, then part of the baddie will be off the edge of the window and not visible.

The bottom edge of the baddie should be just above the top edge of the window. The Y- tom edge there, set the top edge to .

"

for the third and fourth argument.

The rate of speed that the baddie moves down the screen is set in the key. Set it to a random integer between and .

Line 138 will add the newly created baddie data structure to the list of baddie data structures. The program will use this list to check if the player has collided with any of the baddies, and to know where to draw baddies on the window.

Moving the Player's Character

The four movement variables , , and are set to and when Pygame generates the and events, respectively.

(which is the left edge of the window), then should be moved to the left.

object by the number of pixels in . To get

If the reverse cheat is activated, then the baddie should move up by five pixels. Passing for the second argument to will move the Rect object upwards by five pixels.

If the slow cheat has been activated, then the baddie should move downwards, but only by the

stored in the w cheat is activated.

Removing the Baddies

Any baddies that fell below the bottom edge of the window should be removed from the list. Remember that while iterating through a list, do not modify the contents of the list by adding or removing items. So instead of iterating through the list with the loop, iterate through a copy of the list. This copy is made with the blank slicing operator .

The loop on line 163 uses a variable for the current item in the iteration through .

. is the current baddie data structure from the list. Each baddie data structure in the list is a dictionary with a key, which stores a Rect object. So is the Rect object for the baddie.

Finally, the attribute is the Y-coordinate of the top edge of the rectangular area. Remember that the Y-coordinates increase going down. So will check if the top edge of the baddie is below the bottom of the window.

If this condition is , then line 165 removes the baddie data structure from the list.

Drawing the Window

"

image functions. Because the game loop is executed several times a second, drawing the baddies and player in new positions makes their movement look smooth and natural.

First, before drawing anything else, line 168 blacks out the entire screen to erase anything drawn on it previously.

Remember that the Surface object in is the special Surface object because it was the one returned by . Therefore, anything drawn on that Surface object will appear on the screen after is called.

Drawing the Player's Score

Lines 171 and 172 render the text for the score and top score to the top left corner of the window. The expression uses string interpolation to insert the value in the variable into the string.

Pass this string, the Font object stored in the variable, the Surface object on which to draw the text on, and the X- and Y-coordinates of where the text should be placed. The will handle the call to the and methods.

For the top score, do the same thing. Pass for the Y-coordinate instead of so that the top score text appears beneath the score text.

Drawing the Player's Character

The information about the player is kept in two different variables. is a Surface character.

The) on at the location in .

loop draws every baddie on the object. Each item in the and keys contain the Surface object with the baddie image and the Rect object with the position and size information, respectively.

Now that everything has been drawn to , draw this Surface object to the screen by calling .

Collision Detection

Lines 184 checks if the player has collided with any baddies by calling . This function will return list. Otherwise, the function will return .

"

score is greater than it. Then execution will move to line 191.

To keep the computer from running through the game loop as fast as possible (which would be much too fast for the player to keep up with), call to pause for a brief amount of time. The pause will be long enough to ensure that about (the value stored inside the variable) iterations through the game loop occur each second.

The Game Over Screen

"

sound effect. Line 192 calls the function in the module to stop the background music. Line 193 calls the method on the Sound object stored in

.

Lines 195 and 196 call the

object. Line 197 calls to draw this Surface object to the screen. After displaying this text, the game stops until the player presses a key by calling the

function.

After the player presses a key, the program execution will return from the

call on line 198. Depending on how long the player takes to press

"

before a new game starts, line 200 calls .

Modifying the Dodger Game

it for our graphical game. You may find that the game is too easy or too hard. But the game is easy to modify because we took the time to use constant variables instead of typing in the values directly. Now all we need to do to change the game is modify the value set in the constant variables.

For example, if you want the game to run slower in general, change the variable on line 8 to a smaller value such as

since the game loop will only be executed times a second instead of .

If you just want to slow down the baddies and not the player, then change to a smaller value such as . This will make all the baddies move between 1 (the value in) and 4 pixels per iteration through the game loop instead of 1 and 8.

If you want the game to have fewer but larger baddies instead of many fast baddies, then increase to , to , and to . Now that baddies are being added every 12 iterations through the game loop instead of every 6 iterations, there will be half as many baddies as before. But to keep the game interesting, the baddies are now much larger than before.

While the basic game remains the same, you can modify any of the constant variables to drastically affect the behavior of the game. Keep trying out new values for the constant variables until you find a set of values you like the best.

Summary

Unlike our previous text-based games, Dodger really looks like the kind of modern computer game we usually play. It has graphics and music and uses the mouse. While Pygame provides functions and data types as building blocks, it is you the programmer who puts them together to create fun, interactive games.

And it is all because you know how to instruct the computer to do it, step by step, line by line.
" mber crunching and
"
programming. (And still much more to learn!)

Here are several websites that can teach you more about programming Python:

- http://reddit.com/r/inventwithpython This site has several users who could help you with the material in this book.
- http://inventwithpython.com - "
these programs and additional information. This site also has the image and sound files used in the Pygame programs.
- http://inventwithpython.com/pygame My second book, *Making Games with Python & Pygame*, which covers Pygame in more detail. and has the source code for many more games.
- http://inventwithpython.com/hacking My third book, *Hacking Secret Ciphers with Python*, which covers more cryptography and code breaking programs.
download.
- http://inventwithpython.com/automate My fourth book, *Automate the Boring Stuff with Python*, which teaches you practical programming skills.

- http://python.org/doc/ - More Python tutorials and the documentation of all the Python modules and functions.
- http://pygame.org/docs/ - Complete documentation on the modules and functions for Pygame.
- al@inventwithpython.com - My email address. Feel free to email me your questions about this book or about Python programming.

Or you can find out more about Python by searching the web. Go to http://google.com and search

Python programming.

Now get going and invent your own games. And good luck!