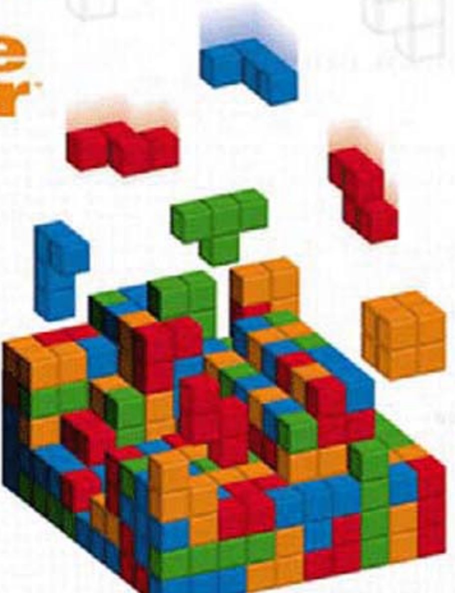
WEB DOWNLOADS AVAILABLE

More Python® Programming

for the absolute beginner™

NO EXPERIENCE REQUIRED



"This series shows that it's possible to teach newcomers a programming language and good programming practices without being boring."

—Lou Grinzo, reviewer for Dr. Dobb's Journal

JONATHAN S. HARBOUR

More Python® Programming for the Absolute Beginner

Jonathan S. Harbour

Course Technology PTR
A part of Cengage Learning

COURSE TECHNOLOGY
CENGAGE Learning™

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

COURSE TECHNOLOGY
CENGAGE Learning™

More Python® Programming for the Absolute Beginner: Jonathan S. Harbour

Publisher and General Manager, Course Technology PTR: Stacy L. Hiquet

Associate Director of Marketing: Sarah Panella

Manager of Editorial Services: Heather Talbot

Marketing Manager: Mark Hughes

Senior Acquisitions Editor: Mitzi Koontz

Project Editor: Jenny Davidson

Technical Reviewer: Keith Davenport

Interior Layout Tech: Value Chain Software India

Cover Designer: Mike Tanamachi

Indexer: Broccoli Information Services

Proofreader: Michael Beady

Library of Congress Control Number: 2011933243

ISBN-13: 978-1-4354-5980-9
ISBN-10: 1-4354-5980-6
eISBN-10: 1-4354-5981-4

Course Technology, a part of Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

Printed in the United States of America
1 2 3 4 5 6 7 13 12 11

This book is dedicated to lone game developers on forums like The Game Programming Wiki (www.gpwiki.org) who put enormous passion into their creative works of interactive fiction, often without recognition. Do what you love and share it with the world!

Acknowledgments

I offer my gratitude to those involved in getting this book finished and into print, especially Mitzi Koontz, Jenny Davidson, Keith Davenport, Mike Tanamachi, and Michael Beady. I would like to say, I hope it sells a million copies so I *never have to work with any of you again*, but that sentiment might be misunderstood and considered rude! So I will just say *thanks*!

About the Author

J**onathan S. Harbour** has been programming since the 1980s. His first game system was an Atari 2600, which he disassembled on the floor of his room as a kid. He has written on C++, C#, Basic, Java, DirectX, Allegro, Lua, DarkBasic, Game Boy Advance, Pocket PC, and game consoles. His other recent books include *Beginning Java SE 6 Game Programming, Third Edition*; *XNA Game Studio 4.0 for Xbox 360 Developers*; *Multi-Threaded Game Engine Design;* and an earlier book in this series, *Visual Basic .NET Programming for the Absolute Beginner* (2003). He holds a master's degree in information systems. Visit him on the web at www.jharbour.com and www.facebook.com/jharbourcom.

Table of Contents

Chapter 1 **Python Has Class** 1

Examining the Geometry Program 2
Getting Started with Python 2
  Python Tools 3
  Python Language 7
Objects in Python 7
  What Came Before? 7
  What's Coming Next? 11
  OOP: The Python Way 13
  Single Inheritance 16
  Multiple Inheritance 17
Summary 19

Chapter 2 **Getting Started with Pygame: The Pie Game** 21

Examining The Pie Game 22
Using Pygame 22
  Printing Text 23
  Looping 24
  Drawing Circles 25
  Drawing Rectangles 27
  Drawing Lines 28
  Drawing Arcs 30
The Pie Game 31
Summary 34

Chapter 3 **File I/O, Data, and Fonts: The Trivia Game** 35

Examining The Trivia Game 35
Python Data Types 36
  More Printing 37
  Getting User Input 39
  Handling Exceptions 40
  The Mad Lib Game 40
File Input/Output 43
  Working with Text Files 43

Working with Binary Files 45
The Trivia Game 47
Printing Text with Pygame 48
The Trivia Class 48
Loading the Trivia Data 49
Displaying the Question and Answers 50
Responding to User Input 52
Going to the Next Question 53
Main Code 54
Summary 55

Chapter 4 **User Input: The Bomb Catcher Game 57**

Examining The Bomb Catcher Game 58
Pygame Events 58
Real-Time Event Loop 59
Keyboard Events 60
Mouse Events 61
Device Polling 61
Polling the Keyboard 61
Polling the Mouse 65
The Bomb Catcher Game 67
Summary 71

Chapter 5 **Math and Graphics: The Analog Clock Demo 73**

Examining The Analog Clock Demo 74
Basic Trigonometry 74
Circle Theory 75
Traversing a Circle’s Perimeter 79
Circle Demo 81
The Analog Clock Demo 83
Getting the Time 84
Drawing the Clock 84
Summary 92

Chapter 6 **Bitmap Graphics: The Orbiting Spaceship Demo . . 93**

Examining The Orbiting Spaceship Demo 93
Using Bitmaps 94
Loading a Bitmap 94
Drawing the Background 95
Drawing the Planet 97

Drawing the Spaceship 98
Orbiting the Planet 101
Orbiting 101
Rotating 103
Summary 107

Chapter 7 **Animation with Sprites: The Escape the Dragon Game 109**

Examining The Escape The Dragon Game 109
Using Pygame Sprites 110
Custom Animation 110
Loading a Sprite Sheet 112
Changing the Frame 112
Drawing One Frame 114
Sprite Groups 115
MySprite Class 116
Sprite Animation to the Test 118
The Escape the Dragon Game 119
Jumping 120
Colliding 121
Source Code 122
Summary 125

Chapter 8 **Sprite Collision Detection: The Zombie Mob Game 127**

Examining The Zombie Mob Game 127
Collision Detection Techniques 128
Rectangle Collision Between Two Sprites 128
Circle Collision Between Two Sprites 129
Pixel-Perfect Masked Collision Between Two Sprites 130
Rectangle Collision Between a Sprite and a Group 130
Rectangle Collision Between Two Groups 131
The Zombie Mob Game 131
Creating Your Own Module 132
Advanced Directional Animation 135
Colliding with Zombies 138
Getting Health 139
Game Source Code 141
Summary 146

Chapter 9 **ARRAYS, LISTS, AND TUPLES: THE BLOCK BREAKER GAME 149**

Examining The Block Breaker Game 149
Arrays and Lists 150
    Lists with One Dimension 150
    Creating a Stack-like List 153
    Creating a Queue-like List 153
    Lists with More Dimensions 154
Tuples 157
    Packing a Tuple 158
    Unpacking a Tuple 158
    Searching for Elements 159
    Counting Elements 159
    Tuples as Constant Arrays 159
The Block Breaker Game 160
    Block Breaker Levels 160
    Loading and Changing Levels 163
    Initializing the Game 165
    Moving the Paddle 165
    Moving the Ball 166
    Hitting the Paddle 167
    Hitting the Blocks 168
    Main Code 169
    MySprite Update 170
Summary 172

Chapter 10 **TIMING AND SOUND: THE OIL SPILL GAME 173**

Examining The Oil Spill Game 173
Sound 174
    Loading an Audio File 174
    Playing an Audio Clip 175
Building The Oil Spill Game 175
    Gameplay 176
    Source Code 179
Summary 183

Chapter 11 **PROGRAM LOGIC: THE SNAKE GAME 185**

Examining The Snake Game 185
Building The Snake Game 186
    Hatching a Snake—the SnakeSegment Class 187
    Raising a Snake—the Snake Class 187

Feeding the Snake—the Food Class 188
Initializing the Game 189
Program Main 191
Growth by Eating Food 193
Biting One's Self Is Not Advisable 194
Falling off the World 195
Teaching the Snake to Move Itself 195
Moving Automatically 196
Getting the Current Direction 197
Moving Toward the Food 197
Other Code Changes 198
Summary 199

Chapter 12 **Trigonometry: The Tank Battle Game 201**

Examining The Tank Battle Game 201
Angular Velocity 202
Calculating Angular Velocity 203
Pygame's Goofy Rotation 203
Moving Forward and Backward at Any Angle 205
Improved Angle Wrapping 206
Building The Tank Battle Game 206
The Tanks 207
The Bullets 211
Main Code 212
Summary 217

Chapter 13 **Random Terrain: The Artillery Gunner Game... 219**

Examining The Artillery Gunner Game 219
Creating the Terrain 220
Defining the Height Map 220
Smoothing the Terrain 226
Locating Grid Points 228
Artillery Cannons 230
Placing the Cannons 230
Drawing the Turrets 231
Firing the Cannons 232
Shots Are A'Flyin 232
Computer Firing 233
Scoring a Hit 234
The Complete Game 235
Summary 244

Chapter 14 **More of Everything: The Dungeon Role-Playing Game........ 245**
Examining The Dungeon Game........ 246
Review of Classic Dungeon RPGs........ 246
Rogue........ 247
NetHack........ 248
AngBand........ 249
Sword of Fargoal........ 251
Kingdom of Kroz........ 251
ZZT........ 251
Creating a Dungeon Level........ 252
Understanding ASCII Characters........ 253
Simulating a Text Console Display........ 257
Generating Random Rooms........ 260
Generating Random Hallways........ 266
Populating the Dungeon........ 272
Adding the Entrance and Exit Portals........ 273
Adding Gold........ 274
Adding Weapons, Armor, and Health Potions........ 276
Adding Monsters........ 277
Complete Dungeon Class........ 278
Adding the Player's Character........ 283
Advanced Gameplay........ 287
Picking Up Items........ 288
Fighting Monsters........ 291
Moving Monsters........ 295
Visibility Range........ 297
Exiting the Level........ 299
Wrapping Up the Gameplay........ 299
Summary........ 304

Appendix A **Installing Python and Pygame........ 305**
Installing Python........ 305
Installing Pygame........ 309

Appendix B **Pygame Key Codes........ 311**

Index........ 317

Introduction

This book continues the study of Python as an introductory text for beginners, following in the footsteps of *Python Programming for the Absolute Beginner, Third Edition* by Michael Dawson. I highly recommend reading that book first if you are an absolute beginner to Python! You will learn the language quickly with Dawson's easy-to-follow examples; clear, concise pacing; and important concepts. While Dawson's book will get you started and get you airborne, so to speak, the book you now hold takes Python into the stratosphere! We will be learning about the awesome Pygame library, which, if you are so inclined, can support OpenGL for advanced 3D rendering! But let's not get too far ahead of ourselves! The primary focus of this book is on using Pygame for 2D graphics, which was only just touched upon in the *final chapter* of our precursor book. *More Python Programming for the Absolute Beginner* begins where *Python Programming for the Absolute Beginner, Third Edition* ended, making the two a complementary pair.

This book teaches most of the important concepts needed to make excellent games with Python. Not merely a "noob" guide, this book delves into complex subjects that will keep you busy for months working with these concepts on your own game ideas. The concepts such as targeting and velocity alone are enough to keep the average programmer busy making many arcade-style shoot-em-up games. These concepts are also found in real-time strategy (RTS) games, since the very same concept used to fire a bullet at a target is used to move a character to a destination. But this book is not just about game programming concepts! We learn the basics first, starting with Python classes, variable data types, text output, lists and tuples, and other important fundamentals of the Python language. The examples start off easy, and by the time you reach the last few chapters you will be using all of the concepts learned to build more complex games—which means you are doing more complex Python programming as well.

If you have not read Dawson's book already, and are a complete newcomer to programming in general, you may have a somewhat difficult time understanding all of the code in this book. That is because this book was intended to be a follow-up of Dawson's book, and as such, it does not stand on its own as a book "for the Absolute Beginner," despite the title. The "More" at the beginning of the title is the key. If you already have *some* programming experience, though, in another language like C++ or Java or C#, then you should be able to get through this book just fine.

This book is based on Python 3.2 and Pygame 1.9. The source code will not compile with earlier versions of Python.

Chapters

Following is a list of the chapters in this book with a short description of each.

1. Python Has Class

This chapter provides a quick overview of the Python language with an emphasis on object-oriented programming. While showing how to create classes with constructors, methods, and properties, this chapter will demonstrate the concepts with a sample program with several classes related to geometry.

2. Getting Started with Pygame: The Pie Game

This chapter contains an introduction to the Pygame library, which will be used in all future chapters. Pygame makes it possible to write graphics demos and games with Python, with 2D shapes and bitmaps.

3. File I/O, Data, and Fonts: The Trivia Game

This chapter teaches how to read and write data with file input/output functions. Sample code shows how to open a file for reading and writing text and binary data. The file access code is then used to make a Trivia Game with graphical text output using Pygame's font support.

4. User Input: The Bomb Catcher Game

This chapter covers user input with Pygame, which is both event-driven and polled. What this means is, we can respond to user input events or we can ask Pygame whether there is user input data. To demonstrate user input, we create a real-time game called Bomb Catcher.

5. Math and Graphics: The Analog Clock Demo

This chapter delves into the complex subjects of math and graphics—that is, using math to produce interesting special effects with graphics. The sample program shows how to make an analog clock with real moving hands, using math to rotate the hands.

6. Bitmap Graphics: The Orbiting Spaceship Demo

This chapter is our first exploration into the world of bitmap graphics. Bitmaps can be created in memory, but are usually loaded from a bitmap file, and used as the artwork in a game. We will use bitmaps to create a chapter example of a spaceship orbiting a planet.

7. Animation with Sprites: The Escape the Dragon Game

This chapter goes much further into advanced bitmapped graphics programming by introducing Pygame's sprite support. We use this awesome feature to create our own sprite class capable of frame animation, and demonstrate it with a sample game that features sprite animation.

8. Sprite Collision Detection: The Zombie Mob Game

This chapter is also related to sprite programming, showing how to detect when game objects collide on the screen, and how to respond to those collisions. This is the basis for most gameplay. To demonstrate, we create a zombie game.

9. Arrays, Lists, and Tuples: The Block Breaker Game

This chapter examines the very important—if not crucial!—subject of arrays, lists, and tuples, all of which are similar in behavior. The purpose of these is to contain other objects, such as sprites, or something simple like just numbers or names. We use this information to create a game that supports level design by defining the game's levels in a list.

10. Timing and Sound: The Oil Spill Game

This chapter shows how to use Pygame's timing and audio features. These subjects are not necessarily related but are often found together because sound effects in a game are often triggered by events that require timing of one sort or another. We create a sample game called The Oil Spill Game to demonstrate these concepts.

11. Program Logic: The Snake Game

This chapter shows how to create the classic Snake Game as a tool for learning how to write source code for game logic. This subject is a simple form of artificial intelligence. We teach the snake in the sample game how to find food on its own without user input.

12. Trigonometry: The Tank Battle Game

This chapter returns to the overall subject of *math* to show how trigonometry gives us powerful tools for game programming. We use several trigonometry functions to make The Tank Battle Game where the tank's turret rotates to follow the mouse cursor on the screen, and this is used for targeting the enemy tank.

13. Random Terrain: The Artillery Gunner Game

This chapter explores the rather complex subject of generating random terrain for The Artillery Gunner Game as the chapter project. We return to vector graphics, rather than using bitmaps, to generate a random landscape, place two opposing artillery guns on it, and then allow the player to take on the computer in trying to blow each other up. This draws on all of the math we have learned and the game is quite fun.

14. More of Everything: The Dungeon Role-Playing Game

This final chapter is a monumental project that shows in a single go how to make a complete role-playing game!

Appendix A: Installing Python and Pygame

This appendix has instructions on how to install Python and Pygame.

Appendix B: Pygame Key Codes

This appendix contains a list of key codes used in Pygame.

Book Resources

The resource files that accompany this book are available for download online. This affords us the benefit of being able to update the resource files at any time, while a more traditional CD-ROM is "set in stone," so to speak. Plus, if you are a serious developer, downloading the files online is actually *faster* than inserting a CD-ROM, copying the files to your hard drive, and so on.

The files may be downloaded from this location: www.courseptr.com/downloads. Please note that you will be redirected to the Cengage Learning site. From here, you may search by book title, ISBN, or author name to receive a list of book resource links.

Alternatively, you may download the book's resource files from the author's website at www.jharbour.com/forum or the author's Facebook page at www.facebook.com/jharbourcom. If you cannot find the source code files for any reason, just post a message on the Facebook page.

Styles

The following styles will be used in this book to highlight facts and concepts that are important for the reader to know about in any given chapter.

HINT

This is what a Hint looks like. Hints offer additional information or suggestions on the current topic.

TRAP

This is what a Trap looks like. Traps recommend ways around problem areas that may help the reader.

TRICK

This is what a Trick looks like. Tricks are meant to give the reader an additional way of accomplishing a task that they may find useful.

The Real World

This is a "In The Real World" callout. This style gives the reader some real-world context that might make the subject seem more relevant.

CHAPTER 1

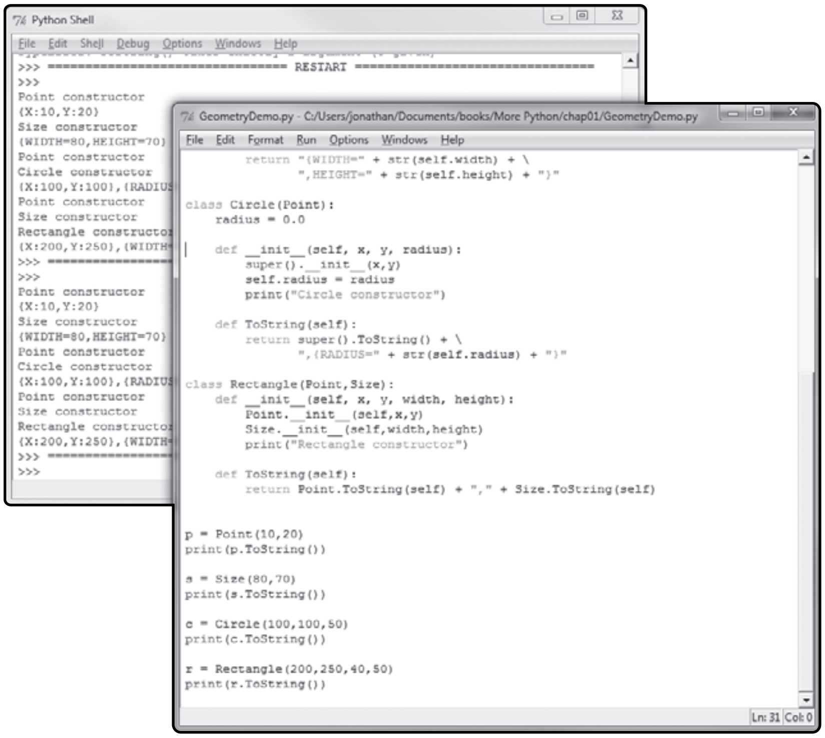
Python Has Class

This chapter will read like a whirlwind tour of Python, hitting on primarily object-oriented programming and making sense of the somewhat strange-looking syntax of the Python language. Python is a tool *and* a language. The *language* includes the syntax and formatting of code according to the Python standard. The *tool* is a software package included with the Python installation that includes an editor. This is some pretty heavy-hitting material for a first chapter. If this is your first exposure to programming in Python, don't let the pacing of this first chapter throw you off—we cover some important details right away, but the book does not get harder with each new chapter. Here is what you will learn:

- How to enter Python code into the IDLE editor
- Using the tools that come with Python
- Reviewing the language features of Python
- Perusing the history of programming languages
- Speculating on the next generation programming methodology
- Polymorphism and inheritance
- Writing an example using multiple inheritance

Examining the Geometry Program

This chapter is a quick romp through Python's object-oriented programming capabilities to get you up to speed on programming in Python the "OOP way" right from the beginning. If you don't understand everything covered in this chapter all at once, don't worry about it, because we will be revisiting all of these concepts in every chapter from here on, while creating games to learn—no, to *master*—the language of Python! Our first example is shown in Figure 1.1.


Python Shell
File Edit Shell Debug Options Windows Help
GeometryDemo.py - C:/Users/jonathan/Documents/books/More Python/chap01/GeometryDemo.py
File Edit Format Run Options Windows Help
Ln: 31 Col: 0

```
>>> ================================ RESTART ================================
>>>
Point constructor
{X:10,Y:20}
Size constructor
{WIDTH=80,HEIGHT=70}
Point constructor
Circle constructor
{X:100,Y:100},{RADIUS
Point constructor
Size constructor
Rectangle constructor
{X:200,Y:250},{WIDTH=
>>> ================
>>>
Point constructor
{X:10,Y:20}
Size constructor
{WIDTH=80,HEIGHT=70}
Point constructor
Circle constructor
{X:100,Y:100},{RADIUS
Point constructor
Size constructor
Rectangle constructor
{X:200,Y:250},{WIDTH=
>>> ================
>>>
```

```
        return "{WIDTH=" + str(self.width) + \
               ",HEIGHT=" + str(self.height) + "}"

class Circle(Point):
    radius = 0.0

    def __init__(self, x, y, radius):
        super().__init__(x,y)
        self.radius = radius
        print("Circle constructor")

    def ToString(self):
        return super().ToString() + \
               ",{RADIUS=" + str(self.radius) + "}"

class Rectangle(Point,Size):
    def __init__(self, x, y, width, height):
        Point.__init__(self,x,y)
        Size.__init__(self,width,height)
        print("Rectangle constructor")

    def ToString(self):
        return Point.ToString(self) + "," + Size.ToString(self)


p = Point(10,20)
print(p.ToString())

s = Size(80,70)
print(s.ToString())

c = Circle(100,100,50)
print(c.ToString())

r = Rectangle(200,250,40,50)
print(r.ToString())
```

Figure 1.1

The Geometry Demo is a quick jaunt through Python's object-oriented programming capabilities.

Getting Started with Python

Python is both a package of software tools and a language. The Python software package includes an editor called IDLE, which is short for... actually, nothing. Idle is the name of a man, not an acronym for integrated development... something, although that seems to fit. The man's name, for those with a penchant for useless trivia, is Eric Idle, one of the founding members of Monty Python, which is where the name comes from: an homage to a British TV show. The Python language is strange too, so it is an appropriate name. Strange in a *beloved* way, of course! If you are completely new to Python and have not read the introductory book by Michael Dawson (*Python Programming for the Absolute Beginner*), then you may be pleasantly surprised to find Python is unlike any other programming language! That makes learning Python a bit of a challenge, but a rewarding one nonetheless.

HINT

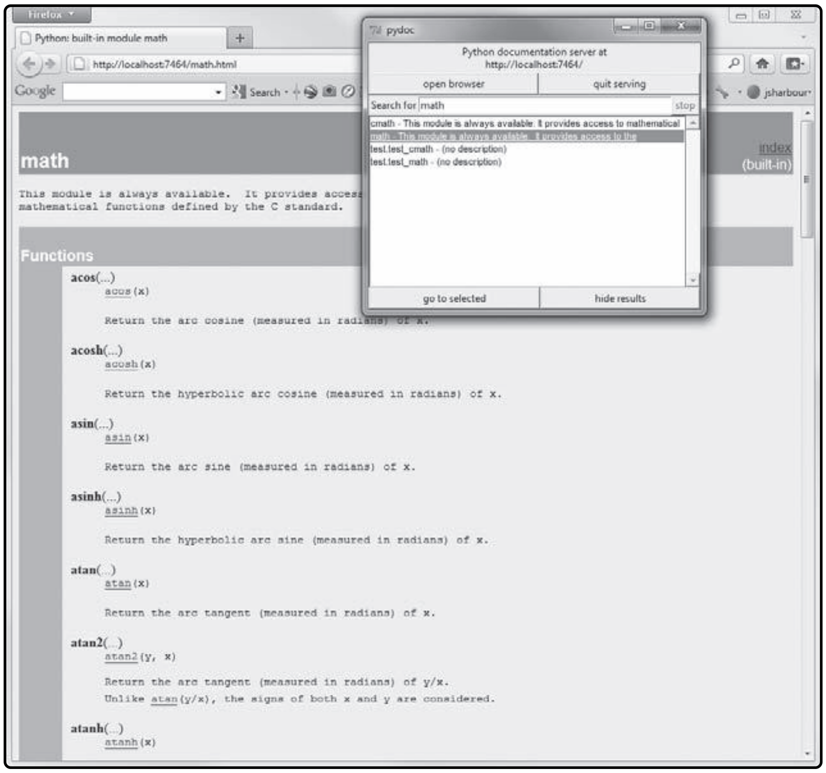
Go to http://www.python.org if you want to download the latest Python package for your operating system of choice.

Python Tools

The Python package includes the Python interpreter and runtime libraries as one would expect, but it also includes several useful utilities that we will take a look at now.

Module Docs (Pydoc)

The Python package differs depending on the operating system, but most commonly the package will include Pydoc, the Python documentation tool. This tool is a small search utility that will locate items in the Python documentation, present the search results in a list, and then bring up any one item in the default web browser. In the Python program group, this utility is also called Module Docs. See Figure 1.2.



Python documentation server at
http://localhost:7464/

Search for math

```
cmath - This module is always available. It provides access to mathematical
math - This module is always available. It provides access to the
test.test_cmath - (no description)
test.test_math - (no description)
```

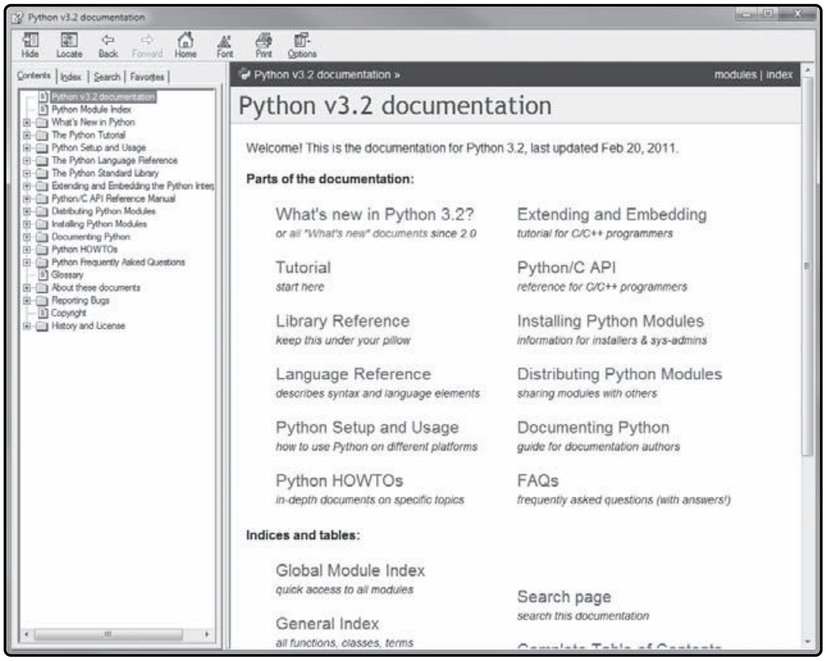
```
math

This module is always available.  It provides access
mathematical functions defined by the C standard.

Functions
acos(...)
    acos(x)

    Return the arc cosine (measured in radians) of x.

acosh(...)
    acosh(x)

    Return the hyperbolic arc cosine (measured in radians) of x.

asin(...)
    asin(x)

    Return the arc sine (measured in radians) of x.

asinh(...)
    asinh(x)

    Return the hyperbolic arc sine (measured in radians) of x.

atan(...)
    atan(x)

    Return the arc tangent (measured in radians) of x.

atan2(...)
    atan2(y, x)

    Return the arc tangent (measured in radians) of y/x.
    Unlike atan(y/x), the signs of both x and y are considered.

atanh(...)
    atanh(x)
```

FIGURE 1.2

Pydoc brings up help pages in the default web browser.

Python Manuals (Pyhelp)

Also found in the program menu is an option, Python Manuals, that brings up the Python documentation in a Windows help file format, shown in Figure 1.3. This version of the documentation is searchable, but may not be as fast a way to look up what you need to find.



Python v3.2 documentation
Python Module Index
What's New in Python
The Python Tutorial
Python Setup and Usage
The Python Language Reference
The Python Standard Library
Extending and Embedding the Python Inter
Python/C API Reference Manual
Distributing Python Modules
Installing Python Modules
Documenting Python
Python HOWTOs
Python Frequently Asked Questions
Glossary
About these documents
Reporting Bugs
Copyright
History and License

Python v3.2 documentation

Welcome! This is the documentation for Python 3.2, last updated Feb 20, 2011.

Parts of the documentation:

What's new in Python 3.2?
or all "What's new" documents since 2.0

Tutorial
start here

Library Reference
keep this under your pillow

Language Reference
describes syntax and language elements

Python Setup and Usage
how to use Python on different platforms

Python HOWTOs
in-depth documents on specific topics

Extending and Embedding
tutorial for C/C++ programmers

Python/C API
reference for C/C++ programmers

Installing Python Modules
information for installers & sys-admins

Distributing Python Modules
sharing modules with others

Documenting Python
guide for documentation authors

FAQs
frequently asked questions (with answers!)

Indices and tables:

Global Module Index
quick access to all modules

General Index
all functions, classes, terms

Search page
search this documentation

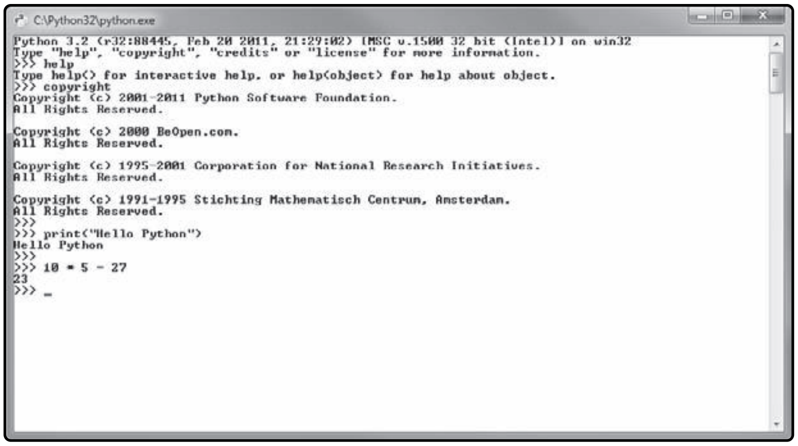
FIGURE 1.3
Python documentation displayed as a Windows help file.

Python (command line)

Python is an interpreted language, which means code is not compiled into an executable file; it is just *interpreted* on the fly in real time. That real time nature includes the Python command prompt, which can accept Python commands one line at a time. Of course, this is a limited way to write Python code and may be thought of as just a parser rather than "code," so to speak. Figure 1.4 shows the command prompt.

IDLE (Python GUI)

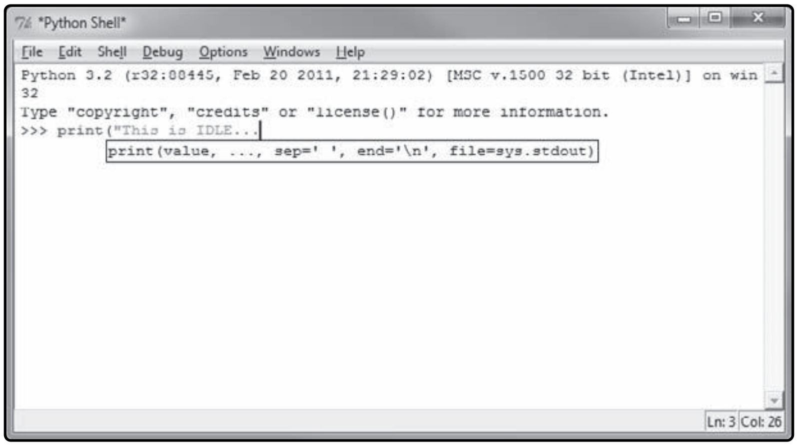
IDLE is a text editor and simple development environment for Python programming. Figure 1.5 shows IDLE in action, displaying a pop-up help message for the code currently being typed in. In this case, it is the syntax for the `print()` function. But this is not the IDLE *editor*; it is just the IDLE command prompt. Yes, we can run an independent prompt like the one shown previously in Figure 1.4, or use the one built into IDLE. To begin actual *editing,* use the File menu and choose New Window, as shown in Figure 1.6. This creates a new source code editor window, shown in Figure 1.7.



```
Python 3.2 (r32:88445, Feb 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> help
Type help() for interactive help, or help(object) for help about object.
>>> copyright
Copyright (c) 2001-2011 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
>>>
>>> print("Hello Python")
Hello Python
>>>
>>> 10 * 5 - 27
23
>>> _
```

FIGURE 1.4

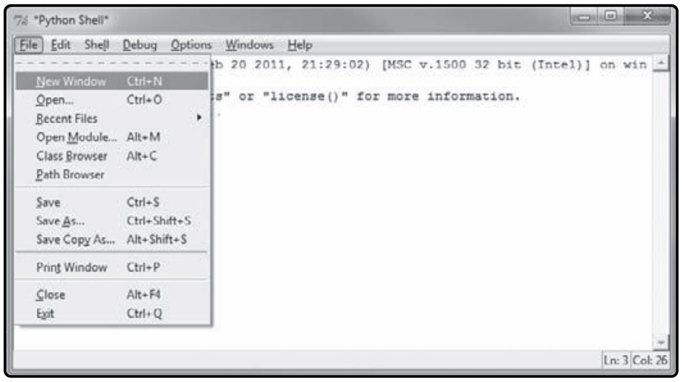
The Python command prompt will interpret commands.



```
Python 3.2 (r32:88445, Feb 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> print("This is IDLE...
          print(value, ..., sep=' ', end='\n', file=sys.stdout)
```

FIGURE 1.5

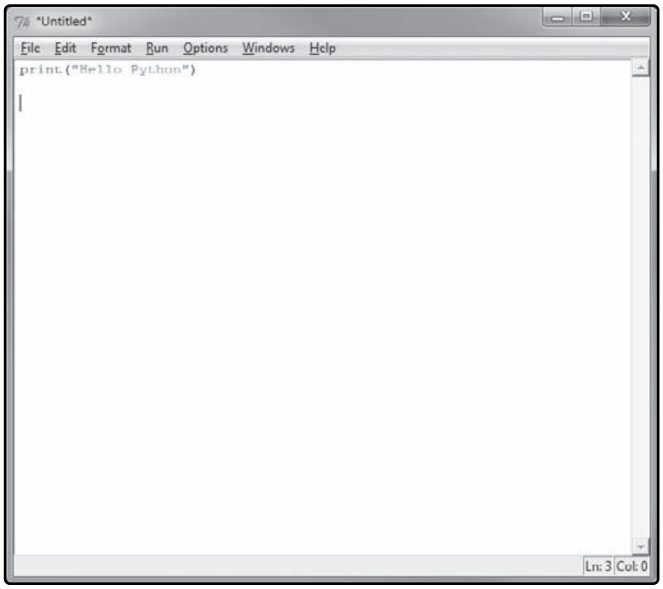
IDLE is the text editor included with Python.



```
b 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)] on win
s" or "license()" for more information.
```

FIGURE 1.6

Creating a new source code editor window with IDLE.

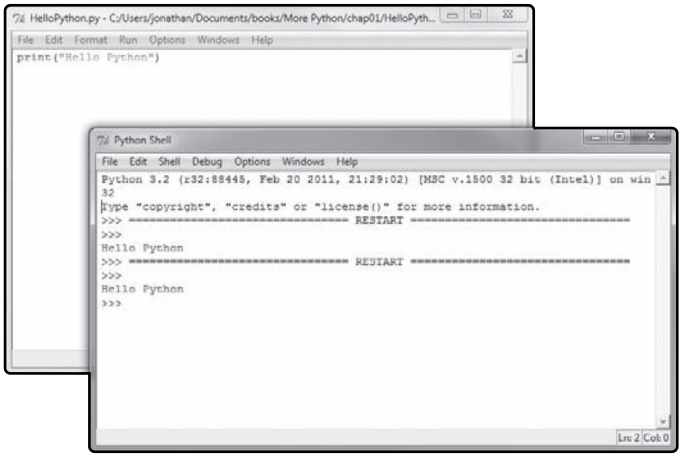


```
print("Hello Python")
```

FIGURE 1.7

Typing in code into the new source code window.

Before doing anything else, you will first want to save the new source code as a file. Until you do this, you cannot have Python run (or interpret) your code. Use the File menu to save the file, then open the Run menu, and choose Run Module. You can also run the code by pressing F5. Now, an interesting thing happens when you run the program. The output goes into the main IDLE window that originally came up! See Figure 1.8. You should leave the prompt (also called the Python Shell) open when editing files because it is the main output window for running programs, even when using a graphical window with Pygame (more on that in the next chapter).


HelloPython.py - C:/Users/jonathan/Documents/books/More Python/chap01/HelloPyth...
File Edit Format Run Options Windows Help
Python Shell
File Edit Shell Debug Options Windows Help
Ln: 2 Col: 0

```
print("Hello Python")
```

```
Python 3.2 (r32:88445, Feb 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello Python
>>> ================================ RESTART ================================
>>>
Hello Python
>>>
```

FIGURE 1.8

Typing code into the new source code window.

Python Language

Python, the *language*, is one very strange-looking programming language that appears to have been designed by a travelling drama troupe with a penchant for obscure Isles humor of the sort that Americans find insufferable and indecipherable. Of course, that is only an emotionally charged, higher education–borne opinion, so one is advised to take it with a grain of salt. Python is also *powerful* and *versatile*, and will surprise you with its capabilities—as soon as you become familiar with it.

It's really surprisingly difficult to compare Python to a language like C++, for there are no opening and closing braces or recognizable function names. The constructor for a Python class is ... well, I don't want to frighten you right off the bat, and have you running back to, say, BASIC, but a constructor is rather obtuse looking. Not that there's anything whatsoever wrong with BASIC! I happen to enjoy an especially awesome tool called QB64 (www.qb64.net), which is featured in another book entitled *Video Game Programming for Kids*. IDLE is a very useful text editor included with the Python package, and we will be using it in this book.

HINT

There is an online reference manual for Python located at: http://docs.python.org/reference.

Objects in Python

Python is an object-oriented programming language, which means it supports at least *some* object-oriented programming concepts. We will spend some time going over those concepts now because this is an effective way to write code. Object-oriented programming (OOP) is a methodology. That is, a way of doing things. There have been several large, "umbrella" methodologies in computer science—that is, methodologies that defined the functionality of programming languages. A methodology is important for the industry in order to make our skills transferrable. If every company used their own methodology, then the skills gained while working for that company would be useless at a different organization. Software engineering is a challenging field, and education is costly, so methodologies are good for everyone involved—skilled developers, employers, and educators who teach the concepts.

What Came Before?

If you have a natural curiosity, which is common among talented programmers, then you may be wondering what type of programming came before the object-oriented paradigm. Let's peruse the subject a bit, if only to illustrate why this is so important to go over up front before we have even really started using Python. Let's peruse where we have come from in order to appreciate where we are today in terms of programming technology.

Structured Programming

Before OOP, the methodology was called procedural or structured programming, which implies that procedures and structures were used, and this is the case. Procedures are often called functions, and we still use them today. Yes, even in an OOP program there will be standalone functions, such as `main()`. A function contained inside an object is called a *method*, and this term has replaced the term *function* when it is part of an object. But, functions can still exist outside of an object, and that is a carryover from the previous "age" (methodology). Structures are complex user-defined types (UDTs) that can contain many variables organized together. The most popular structured language was C. But this was a long and successful methodology that some still adhere to today. The time frame for the structured movement was the 1980s and 1990s, but of course there was quite a bit of overlap in both directions. In the electronics industry, many software development kits (SDKs) are still developed in a structured manner, with libraries of functions supplied to control an electronic device (such as a video card or embedded system). It might be argued that the development of the C language (at right around 1970) ushered in structured programming in a major way. The C language was used to create the UNIX operating system.

Here is a quick example of a structured program in Python.

```
# Structured program

# Function definition
def PrintName(name):
    print("The name is " + name + ".")

# Start of program
PrintName("Jane Doe")
```

The program produces this output:

```
The name is Jane Doe.
```

HINT

A comment line in Python begins with the pound character (#).

The function definition begins with the word `def`, followed by the function name, parameters, and a colon. There are no code block characters in Python, like the opening brace (`{`) and closing brace (`}`) in C++. In Python, the end of the function is undefined, assumed to end before the next *non-indented* line. Let's try a little experiment to test the behavior of Python. Here's our example again, without any comment lines. What do you think it will print out?

```
def PrintName(name):
    print("The name is " + name + ".")
print("END")
PrintName("Jane Doe")
```

The output is:

```
END
The name is Jane Doe.
```

This is a surprise to most Python beginners. What's happening is the `print("END")` line is indented on the left, so it became the first line of the program, followed by `PrintName("Jane Doe")`, the second line. The function definition is not considered part of the *main program*, and is only run when the function is called. What happens if we move the function definition below the main program like this?

```
PrintName("Jane Doe")

def PrintName(name):
    print("The name is " + name + ".")
```

That code actually produces a syntax error because the `PrintName` function could not be found. This tells us that Python must parse functions before they are called. In other words, function defs must be "above" code where the function is called.

```
Traceback (most recent call last):
  File "FunctionDemo.py", line 4, in <module>
    PrintName("Jane Doe")
NameError: name 'PrintName' is not defined
```

TRAP

When saving a source code file with IDLE, be sure to include the extension .PY, as IDLE does not automatically append the extension.

Sequential Programming

Structured programming evolved from the earlier sequential programming methodology. This is not a formal textbook description but more of a descriptive one. Sequential programs required line numbers before each line of code. While it was possible to jump (with `goto` or `gosub` commands) to another line of the program, and that was an early evolution in the direction of structure, sequential programs had the tendency to get stuck with a certain level of complexity, beyond which the code became indecipherable, or impossible to change. The problem at the time was called "spaghetti code" because of the way the "flow" of the program seemed to go every which direction. The two most popular sequential languages were BASIC and FORTRAN, and the heyday for these languages was the 1970s and 1980s. As developers grew tired of maintaining spaghetti code, a paradigm shift was needed, and structure was ushered in with the introduction of new structured languages like Pascal and C.

```
10 print "I am freaking out!"
20 goto 10
```

TRICK

Do you actually find this sequential code kind of interesting? I do! It takes me back a few years. There's a great (and free) compiler called QB64 at www.qb64.net, that supports all of the old flavors of BASIC, QBASIC, and QuickBasic (which is structured, not sequential). In addition, QB64 supports OpenGL, so there's potential for advanced graphics and gameplay as well as support for the old-school variations of BASIC.

Mnemonic Programming

Prior to sequential programming, developers wrote code much closer to the level of the computer hardware, using assembly language. An "assembler" program was like a compiler, but it would convert mnemonic ("nee-monic") instructions directly into machine code in an object or binary file, ready to be run by the processor one byte at a time. One assembly mnemonic instruction correlates directly to one machine instruction that the processor understands. This is like speaking the machine's own language and is very challenging! In the old days of MS-DOS, these assembly instructions would change the video mode to a graphics mode with 320x200 resolution and 256 (8-bit) color, which was great for an IBM PC game back in the early 1990s because it was *fast*. Remember, in this time period, there were no video cards like we have today, just "video out" built into the ROM BIOS and whatever modes were supported by an operating system. This was called the infamous "VGA mode 13h" that all game developers loved at the time.

```
mov ax, 13h
int 10h
```

HINT

Here is an interesting historical site dedicated to programming VGA mode 13h: http://www.delorie.com/djgpp/doc/ug/graphics/vga.html.

"AX" is a 16-bit processor register, an actual physical circuit on the processor that was treated like a general-purpose "variable" of sorts, to use a familiar term without the language of electronics engineering. There were three other general-purpose registers: BX, CX, and DX. They were themselves upgrades from the earlier 8-bit Intel processors, which had registers called A, B, C, and D. When 16-bit was developed, these registers were expanded into AL/AH, BL/BH, CL/CH, and DL/DH, which were two 8-bit parts of each 16-bit register. It's not as complicated as it might sound at first. Put a value into one or more of these variable registers, then "launch" a procedure by calling an interrupt. In the case of the VGA mode change, the interrupt was 10h.

In the Real World

If you like this subject—electronics engineering and assembly language—there is a modern counterpart to the jobs of old: device driver programming. It is a black art today, reserved only for those engineers who really understand the hardware. So, you see, studying assembly language could be a very beneficial (and fun) subject to study if it interests you.

What's Coming Next?

Now that we have taken a brief look at the programming methodologies of the past that led up to this point, as a means of *understanding* and *appreciating* the tools and languages we have today, let's talk about the current state of the art and what's coming. Today, object-oriented programming is still the main methodology used by most professional programmers. It is the basis for popular industry-leading tools like Visual Studio and the .NET Framework from Microsoft. The major compiled OOP languages used in business and science today are C++, C# ("see-sharp"), BASIC (the modern variation known as *Visual* Basic), and Java. There are others, but these are the big league players.

Python and LUA are scripting languages. In comparison to compiled languages like C++, Python and LUA are handled quite differently—*interpreted* rather than *compiled*. When you run a Python program (a file with an extension of .PY), it is not compiled, it is run. You could insert syntax errors into a function in Python and unless that function is called, Python will never complain about the errors!

```
# Funny syntax error example

# Bad function!
def ErrorProne():
    printgobblegobble("Hello there!")

print("See, nothing bad happened. You worry too much!")
```

There is no function called `printgobblegobble()` in Python or in this program, so that should have generated an error! Here is the output:

```
See, nothing bad happened. You worry too much!
```

But, if you add a call to the `ErrorProne()` function, this will be the output:

```
Traceback (most recent call last):
  File "ErrorProne.py", line 9, in <module>
    ErrorProne()
  File "ErrorProne.py", line 5, in ErrorProne
    printgobblegobble("Hello there!")
NameError: global name 'printgobblegobble' is not defined
```

Now, there are limits to this apparent *ignorance* on the part of Python. If you blatantly define a variable the wrong way, it will generate an error first before running. There's another weird thing you can do in Python to totally screw things up: using reserved words as variables. Behold:

```
print = 10
print(print)
```

The first line works just fine, but the second line produces this error:

```
Traceback (most recent call last):
  File "ErrorProne.py", line 8, in <module>
    print(print)
TypeError: 'int' object is not callable
```

What this error means is, `print` has become a variable, an integer to be exact, set to the value of `10`. Then we try to call the old `print()` function, and Python doesn't get it. Because the old `print()` function has been bypassed. Now, this strange behavior does not apply to *reserved words* in the Python language, like while, for, if, and so on, only to functions. I think you will be surprised to find a great amount of flexibility in Python as a scripting language.

A traditional compiler, like GCC or Visual C++, would pitch a fit before even thinking about running such code! But then, these are *compilers*. They parse the flow of a program completely before converting it into object code. And therein lies the disadvantage: a compiler cannot work with unknowns, only that which is known, while a scripting language can handle the unknown very well!

The next methodology will evolve from OOP the way sequential evolved into structured, and structured into OOP, with telltale signs of the change showing up in the current methodology before the paradigm change happens. The change happening today to OOP might be called *adaptive programming*. In the fast-paced world of today, no one sits down at their computer with a 200-page manual for WordPerfect or Lotus 1-2-3 like we did in the old days of computing. There are still people who think "read the manual!" is a reasonable answer to technical questions, but today it's rare if a product even comes with a semblance of a manual. Today, systems must be interactive and *adaptable*. The next evolution beyond OOP could very well be entity oriented programming or EOP.

Instead of writing code with objects containing *properties* (variables) and *methods* (functions), imagine using *entities*—self-contained objects that work together with simple rules to solve complex problems. That seems to be the direction of A.I. research, and could very well be adapted into existing OOP languages *today*. In fact, there are early signs of this already happening. Ever heard of web services? A web service is a self-contained object that resides online, and can be used by a program to perform unique services that it did not know how to do. Those web services might just ask for a parameter for an inventory database and return a list of items that match the query. This form of program interaction sure beats writing SQL (structured query language)—the language of relational databases! What about taking it to the next level? Instead of tapping into a known service, what about querying for a service online using some sort of repository or search engine?

As another possible example, imagine an online storehouse of game entities that can be used in a game (most likely pioneered by indie developers or open source teams), where the entity will come with its own art assets (2D sprites, 3D meshes, textures, audio clips, etc.) and it's own behaviors (such as a Python script). An existing game engine that requires assets to be in a certain format could use this sort of EOP concept to extend gameplay. Imagine you are playing a game, some sort of world building game like Minecraft (www.minecraft.net), and you imagine some new character in the game. So, you query for it, "I need a short wooden chair." After a slight delay for the query to be sent out, a short wooden chair appears in front of you in the game. Assuming there is an online repository of game assets for an engine like Minecraft, it's not beyond reason to imagine this sort of development to occur.

OOP: The Python Way

We have done enough historical analysis and speculation to trigger some imaginative thinking, so let's learn about something concrete and practical now—the current OOP methodology as it is implemented in Python. Or, in other words, creating objects in Python! Python does support some OOP features, but not all to the degree of a highly specific language like C++. Let's get the terminology straight first, before we get started. A *class* is a blueprint for an object. A class cannot do anything, because it is a blueprint. An object does not exist until it is created at runtime. So, when we are writing the code, it is a *class definition*, not an object. It is only truly an *object* when it is created at runtime from the *blueprint* of a class. A class *function* is called a *method*. A class *variable* is usually accessed as a property (a sort of method for getting or setting the value of a variable). When an object is created, the class is *instantiated* into the object.

Let's learn about the specifics of Python's OOP features. Here is an example:

```
class Bug(object):
    legs = 0
    distance = 0

    def __init__(self, name, legs):
        self.name = name
        self.legs = legs

    def Walk(self):
        self.distance += 1

    def ToString(self):
        return self.name + " has " + str(self.legs) + " legs" + \
            " and taken " + str(self.distance) + " steps."
```

Every definition must be followed by a colon at the end of the line. The key word *self* describes the current class, and is equivalent to *this* in C++. All class variables must be prefixed with "self." in order to be recognized as members of the class; otherwise, they are treated as local.

The `def __init__(self)` line begins the class constructor—the first method that runs when the class is instantiated. Class variables can be declared and initialized outside of the constructor when they are declared.

Polymorphism

The term *polymorph* means "many forms" or "many shapes," so polymorphism is the ability to take many forms or shapes. In the context of a class, this means we can use methods with many shapes—that is, many different sets of parameters. In Python, we can use optional parameters to make a method more versatile. The constructor of our new Bug class can be transformed with the use of optional parameters like so:

```
def __init__(self, name="Bug", legs=6):
    self.name = name
    self.legs = legs
```

Likewise, the Walk() method can be upgraded to support an optional parameter:

```
def Walk(self,distance=1):
    self.distance += distance
```

Data Hiding (Encapsulation)

Python does not allow variables or methods to be declared as private or protected, as the scope of all things is public in Python. But, if you want to write code that makes it *look* like data hiding is working, that is definitely doable. For instance, this code might be used to access or change the distance variable (which we would *assume* is private, even though it isn't):

```
def GetDistance(self):
   return p_distance

def SetDistance(self, value):
    p_distance = value
```

From a data hiding point of view, you could rename distance to p_distance (making it appear to be a private variable), and then access it using these two methods. That is, if data hiding is important in your program.

Inheritance

Python supports inheritance of base classes. When a class is defined, the base class is included in parentheses:

```
class Car(Vehicle):
```

In addition, Python supports *multiple inheritance*; that is, more than one parent or base class can be inherited from in a child class. For example:

```
class Car(Body,Engine,Suspension,Interior):
```

As long as the variables and methods in each parent class do not conflict with each other, the new child class can access them all without incident. But, if there are any conflicts, the conflicted variable or method is used from the parent that comes first in the inheritance ordering.

When a Python class inherits from a base class, all of the variables and methods of the parent are available. Variables can be used, and methods can be overridden. When calling the constructor or any method of a base class, we can use `super()` to refer to the base:

```
return super().ToString()
```

But when multiple inheritance is involved, the name of the parent class must be used when both share the same variable or method name, to resolve the confusion.

Single Inheritance

Let's look at an example of single-parent inheritance first. Here is a `Point` class and a `Circle` class that inherits from it:

```
class Point():
    x = 0.0
    y = 0.0

    def __init__(self, x, y):
        self.x = x
        self.y = y
        print("Point constructor")

    def ToString(self):
        return "{X:" + str(self.x) + ",Y:" + str(self.y) + "}"

class Circle(Point):
    radius = 0.0

    def __init__(self, x, y, radius):
        super().__init__(x,y)
        self.radius = radius
        print("Circle constructor")

    def ToString(self):
        return super().ToString() + \
               ",{RADIUS=" + str(self.radius) + "}"
```

We can test these classes simply enough:

```
p = Point(10,20)
print(p.ToString())

c = Circle(100,100,50)
print(c.ToString())
```

That produces this output:

```
Point constructor
{X:10,Y:20}

Point constructor
Circle constructor
{X:100,Y:100},{RADIUS=50}
```

We can see that `Point` is simple enough in function, but `Circle` calls the `Point` constructor before its own, and then uses the complex `ToString()` call from `Point` and adds its own new radius property to the mix. This is really helpful to see which is why all of our classes have a `ToString()` method.

TRAP

Multiple inheritance is a quagmire! I recommend avoiding it when possible and just keeping classes simple and straightforward, with perhaps one level of inheritance at most. Give your classes a lot of functionality rather than dividing them up across several classes, for best results.

Now, when the `Circle` class is created, the constructor is called with the three parameters passed to it (100,100,50). Note that the parent (`Point`) constructor is called to handle the x and y parameters, while the radius parameter is handled inside `Circle`:

```
    def __init__(self, x, y, radius):
        super().__init__(x,y)
        self.radius = radius
```

The call to `super()` invokes the constructor of the `Point` class, which is the parent or base class of `Circle`. This works marvelously when single inheritance is used!

Multiple Inheritance

I would be remiss by not at least showing you how multiple inheritance works, even if it is a quagmire! We essentially cannot use `super()` to invoke anything in the parent class when using multiple inheritance, *unless* the variables and methods of each parent class are unique. Here is another pair of classes that build on the previous two already shown. Remember when

I warned you that Python was a strange-looking language? We are now seeing that here! Try to remember that Python is a script language, not a compiled language. Python code is *interpreted* while it runs.

```
class Size():
    width = 0.0
    height = 0.0

    def __init__(self,width,height):
        self.width = width
        self.height = height
        print("Size constructor")

    def ToString(self):
        return "{WIDTH=" + str(self.width) + \
               ",HEIGHT=" + str(self.height) + "}"

class Rectangle(Point,Size):
    def __init__(self, x, y, width, height):
        Point.__init__(self,x,y)
        Size.__init__(self,width,height)
        print("Rectangle constructor")

    def ToString(self):
        return Point.ToString(self) + "," + Size.ToString(self)
```

The `Size` class is a new helper class, while `Rectangle` is our real focus for the example. Here, we are inheriting from both `Point` and `Size`:

```
class Rectangle(Point,Size):
```

`Point` was defined earlier, while `Size` was defined just above. Now, we could just begin using `Point.x`, `Point.y`, `Size.width`, and `Size.height`, as well as the `ToString()` methods in each. Python would not complain. But, the idea is to auto-initialize parent classes by calling their constructors. Otherwise, we are losing all benefit of OOP and might as well just write structured code! So, the `Rectangle` constructor must call each parent constructor by name:

```
    def __init__(self, x, y, width, height):
        Point.__init__(self,x,y)
        Size.__init__(self,width,height)
```

Note that x and y are passed to `Point.__init__()`, while width and height are passed to `Size.__init__()`. This properly initializes those variables within their respective classes. Of *course* we could have just defined x, y, width, and height right inside `Rectangle`, but this is a demonstration! In general, I would recommend doing just that to simplify the code! There is absolutely no reason to use inheritance in this manner in a real-life program. It is purely for illustration. Testing out our new `Size` and `Rectangle` classes:

```
s = Size(80,70)
print(s.ToString())

r = Rectangle(200,250,40,50)
print(r.ToString())
```

Produces this output:

```
Size constructor
{WIDTH=80,HEIGHT=70}

Point constructor
Size constructor
Rectangle constructor
{X:200,Y:250},{WIDTH=40,HEIGHT=50}
```

Now this is really quite interesting! `Size` is simple enough to follow, but look at the output for `Rectangle`! We have a call to the `Point` constructor and `Size` constructor, exactly as planned! Furthermore, the `ToString()` method combines the output of `Point.ToString()` and `Size.ToString()` effectively.

Summary

This chapter was very fast paced for the first chapter on Python programming! Is your hair messy from going so fast? Don't worry, we're going to put code like this to use in a practical way, by actually drawing points, circles, and rectangles, among other things! We will also create a sprite class for drawing game characters on the screen with animation as a learning tool for Python! The good news is, this was probably the hardest chapter because it was your first exposure to not only the strange-looking Python syntax, but also to object-oriented programming in all likelihood! As you will find in later chapters, the straightforward approach to learning a programming language is usually the best way. I hope you are ready to go, because in the very next chapter we'll begin learning Pygame!

CHALLENGES

1. **Open the GeometryDemo.py program and create your own new class that inherits from `Point`, called `Ellipse`, with a horizontal *and* vertical radius rather than just a single radius like the one in `Circle`.**
2. **Add a new method to the `Rectangle` class called `CalcArea()`, that returns the area of the `Rectangle`. The formula is: Area I Width * Height. Test the method to be sure it works.**
3. **Add a new method to the `Circle` class called `CalcCircum()`, that returns the circumference around the circle. The formula is: Circumference I 2 * Pi * Radius (where Pi I 3.14159). Test the method to be sure it works.**

CHAPTER 2

Getting Started with Pygame: The Pie Game

This chapter introduces a game library called Pygame that was developed to make it possible to draw graphics, get user input, do animation, and use a timer to make the game run at a consistent frame rate. We're going to just get started with Pygame in this chapter, learn the basics of drawing shapes and text, and will be writing quite a bit of code along the way. As you will see, Pygame does more than just provide drawing functions for shapes and bitmaps. Pygame also provides services for getting user input, handling audio playback, and polling the mouse and keyboard. We will get to these additional topics in due time.

Here are the topics covered in this chapter:

- Using the Pygame library
- Printing text with fonts
- Using looping to repeat actions
- Drawing circles, rectangles, lines, and arcs
- Creating The Pie Game

Examining the Pie Game

Our example in this chapter is called The Pie Game. The Pie Game uses Pygame to draw filled-in pie slices. To draw a pie slice in our Pie Game with Pygame, the user presses number keys corresponding to the pie pieces. We then use Pygame drawing functions to draw the pie pieces. The player wins by pressing the keys for all of the pieces without making a mistake.

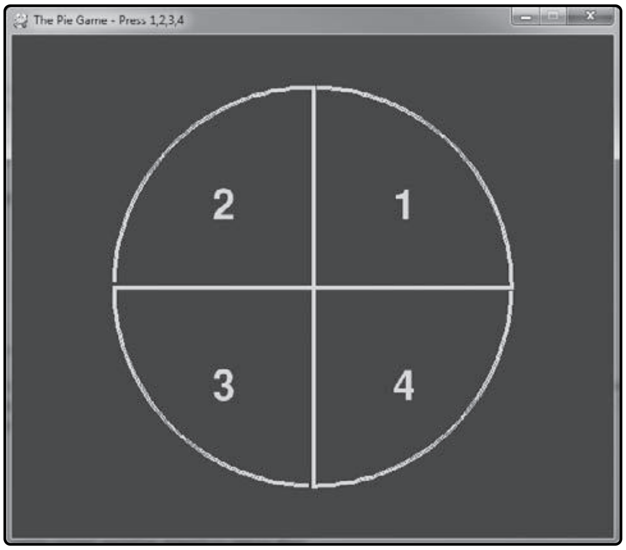


Figure 2.1

The Pie Game.

TRAP

Pygame must be installed before you can use it, because Pygame is not packaged with Python. Download Pygame from http://www.pygame.org/download.shtml. It is important to get the *right version* of Pygame that goes with the version of Python you are using. This book uses Python 3.2 with Pygame 1.9. If you need help getting it installed, please see Appendix A for more details.

Using Pygame

The first step to using Pygame is to import the Pygame library into our Python program so it can be used.

```
import pygame
```

The next step is to import all of the constants in Pygame so they are more readily accessible in our code. This is *optional* but tends to make the code cleaner. Some Python programmers dislike importing everything from a library due to efficiency concerns, but this makes our code a *whole* lot easier to read.

```
from pygame.locals import *
```

Now we can initialize Pygame:

```
pygame.init()
```

Now that Pygame has been initialized, we have access to all of the resources of the library. The next order of business is to gain access to the display system and create a window. The resolution is up to you, but did you note that the screen width and height parameters are enclosed in parentheses? The (600,500) pair becomes a point with an x and y property. In Python, source code syntax is loosely enforced by the interpreter, so we can write code like this where a more strongly typed language—such as C++—would not allow it.

```
screen = pygame.display.set_mode((600,500))
```

HINT

An excellent Pygame reference manual is found online at: http://www.pygame.org/docs/index.html.

Printing Text

Pygame supports text output to the graphics window using `pygame.font`. To draw text, we must first create a font object:

```
myfont = pygame.font.Font(None,60)
```

The name of a TrueType font can be supplied to the `pygame.font.Font()` constructor, such as "Arial," but using `None` (no quotes) causes the default Pygame font to be used. A point size of 60 is quite large, but this is a simple example. Now, drawing text is not a *light* process with Pygame; it's a *heavy* process. Meaning, text is not just quickly drawn to the screen, it is rendered onto a surface which is then drawn to the screen. Because this is a rather time-consuming process, it is advised to create the text surface (or image) in memory first, and then draw the text as an image. When we simply *must* draw text in real time, that's okay, but if the text doesn't change, it's better to pre-render the text onto an image.

```
white = 255,255,255
blue = 0,0,255
textImage = myfont.render("Hello Pygame", True, white)
```

The `textImage` object will be a surface that can be drawn with `screen.blit()`, our die-hard drawing function that will be used extensively in all of our games and demos! The first parameter is obviously the text message; the second parameter is a flag to anti-alias the font (to improve quality); the third parameter is the color (an RGB value).

To draw the text, the usual process is to clear the screen, do our drawing, and then refresh the display. Let's see all three lines of code:

```
screen.fill(blue)
screen.blit(textImage, (100,100))
pygame.display.update()
```

Now, if you run the program at this point, what happens? Go ahead and give it a try. Did you see the window come up after running the program? Since there is no delay anywhere in our code, the window should come up and then close just as quickly. A delay is needed. But, instead of a delay, we'll go a step further.

Looping

There are two problems with the simplistic example we've just seen. First, it just runs once and then quits. Second, there's no way to get any user input (even if it did not just immediately exit). So, let's look into correcting that oversight. First, we need a loop. This is done in Python with the `while` keyword. The `while` statement will execute the code following the colon until the condition is false. As long as the *while* condition is true, it will keep running:

```
while True:
```

Next, we will create an event handler. At this early stage, all we want to happen is for the window to stay up until the user closes it. The close event can be clicking the "X" at the upper- right corner of the window, or by just pressing any key. Note that the code is indented within the `while` loop. Any code that is indented after this point will be contained in the `while` loop.

```
while True:
    for event in pygame.event.get():
        if event.type in (QUIT, KEYDOWN):
            sys.exit()
```

Lastly, we add the drawing code and screen refresh indented in the `while` loop, and this wraps up the program. Just for the sake of learning, here is the complete program without any blank lines or comments. The output of the program is shown in Figure 2.2.

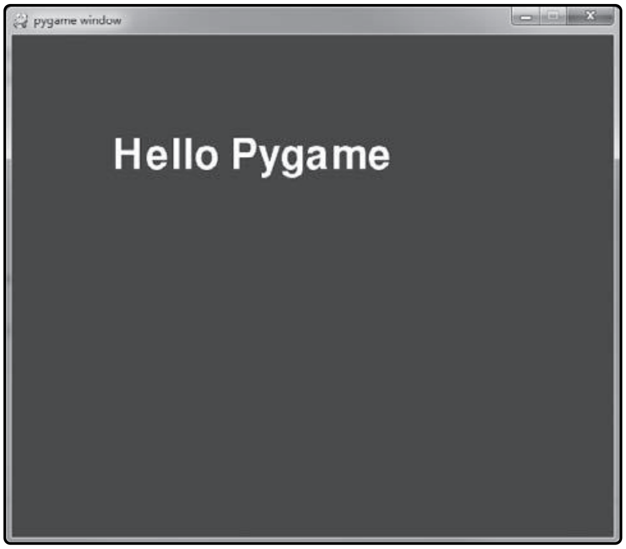


FIGURE 2.2
The Hello Pygame program.

```
import pygame
from pygame.locals import *
white = 255,255,255
blue = 0,0,200
pygame.init()
screen = pygame.display.set_mode((600,500))
myfont = pygame.font.Font(None,60)
textImage = myfont.render("Hello Pygame", True, white)
while True:
    for event in pygame.event.get():
        if event.type in (QUIT, KEYDOWN):
            sys.exit()
    screen.fill(blue)
    screen.blit(textImage, (100,100))
    pygame.display.update()
```

Drawing Circles

We can draw many different shapes with the pygame.draw library. Figure 2.3 shows the circle drawn by the example code shown. To draw a circle, we use `pygame.draw.circle()`, and pass a number of parameters to customize the size, color, and position of the circle.

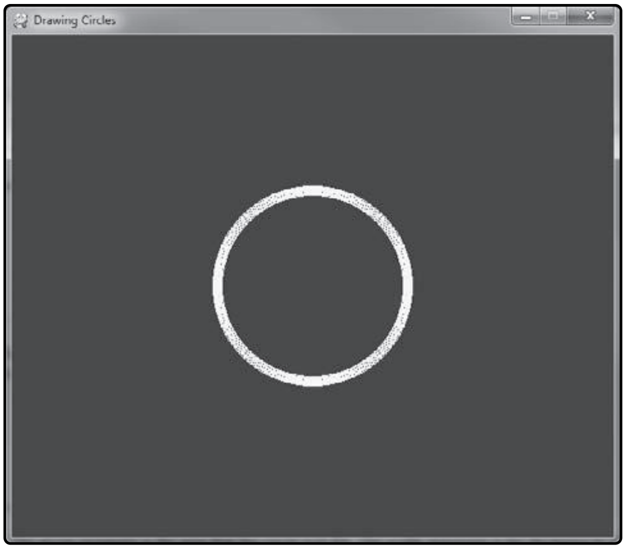


Figure 2.3

The Drawing Circles example.

```
import pygame
from pygame.locals import *
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("Drawing Circles")
while True:
    for event in pygame.event.get():
        if event.type in (QUIT, KEYDOWN):
            sys.exit()

    screen.fill((0,0,200))

    #draw a circle
    color = 255,255,0
    position = 300,250
    radius = 100
    width = 10
    pygame.draw.circle(screen, color, position, radius, width)

    pygame.display.update()
```

Drawing Rectangles

To draw a rectangle, we use the `pygame.draw.rect()` function with a number of parameters. The window displayed by this program is shown in Figure 2.4. This example is a little more advanced than the one before. Instead of just drawing a rectangle at the center of the screen, this example *moves* the rectangle! The way this works is, we keep track of the rectangle's position outside of the `while` loop (with `pos_x` and `pos_y`), and create a pair of velocity variables (`vel_x` and `vel_y`). *Inside* the `while` loop, we can update the position using the velocity, and then some logic keeps the rectangle on the screen. The way this works is, any time the rectangle reaches an edge of the screen, the velocity is reversed!

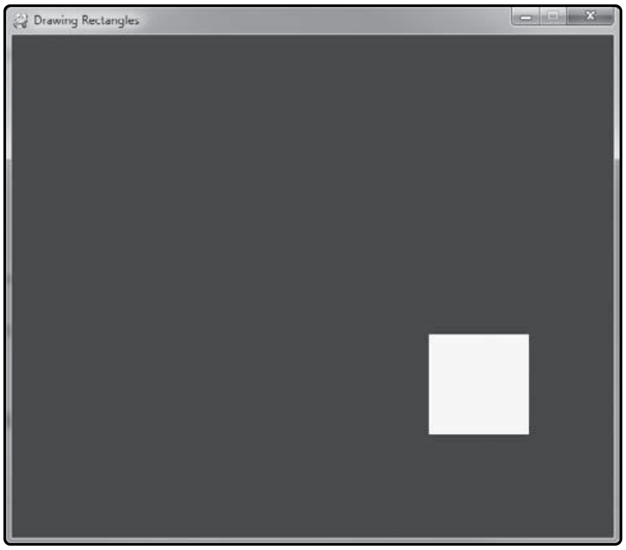


Figure 2.4
The Drawing Rectangles example.

```
import pygame
from pygame.locals import *
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("Drawing Rectangles")

pos_x = 300
pos_y = 250
vel_x = 2
vel_y = 1
```

```
while True:
    for event in pygame.event.get():
        if event.type in (QUIT, KEYDOWN):
            sys.exit()

    screen.fill((0,0,200))

    #move the rectangle
    pos_x += vel_x
    pos_y += vel_y

    #keep rectangle on the screen
    if pos_x > 500 or pos_x < 0:
        vel_x = -vel_x
    if pos_y > 400 or pos_y < 0:
        vel_y = -vel_y

    #draw the rectangle
    color = 255,255,0
    width = 0 #solid fill
    pos = pos_x, pos_y, 100, 100
    pygame.draw.rect(screen, color, pos, width)

    pygame.display.update()
```

Drawing Lines

We can draw straight lines using the `pygame.draw.line()` function. Line drawing is a little more complex than drawing other shapes, only because both the start position and end position of the line must be supplied. We can draw the line in any color and with any desired line width. Figure 2.5 shows the example running.

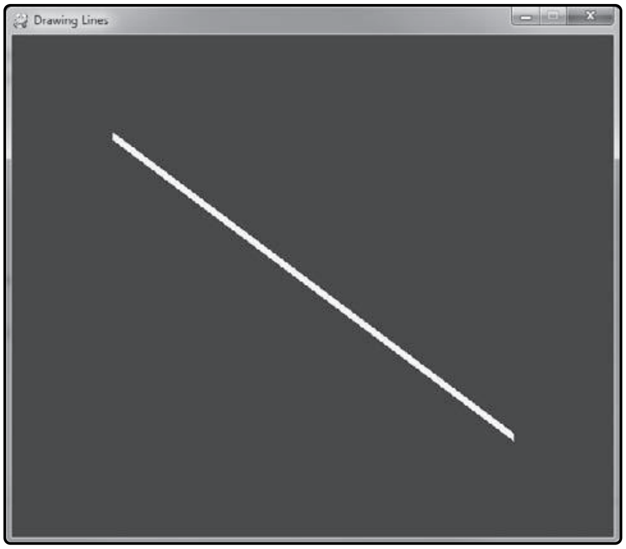


Figure 2.5

The Drawing Lines example.

```
import pygame
from pygame.locals import *
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("Drawing Lines")

while True:
    for event in pygame.event.get():
        if event.type in (QUIT, KEYDOWN):
            sys.exit()

    screen.fill((0,80,0))

    #draw the line
    color = 100,255,200
    width = 8
    pygame.draw.line(screen, color, (100,100), (500,400), width)

    pygame.display.update()
```

Drawing Arcs

An arc is a partial circle that can be drawn with the `pygame.draw.arc()` function. This is another rather complex shape that requires additional parameters. We have to supply a rectangle that represents the boundary of the arc, beginning with the upper-left corner and then the width and height, within which the arc will be drawn. Next, we have to supply the starting angle and ending angle. Normally, we tend to think about angles in terms of degrees, but trigonometry works with radians, and that is the form of circle measurement we must use. To convert an angle to radians, we can use the `math.radians()` function, with the degree angle as the parameter. Since the math library is required, we have to import math at the top of the program. Figure 2.6 shows the output of the example listed below.

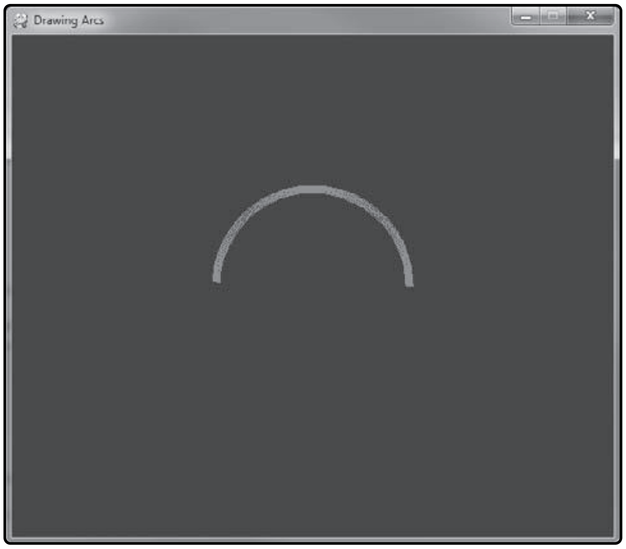


FIGURE 2.6

The Drawing Arcs example.

```
import math
import pygame
from pygame.locals import *
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("Drawing Arcs")

while True:
    for event in pygame.event.get():
        if event.type in (QUIT, KEYDOWN):
            sys.exit()

    screen.fill((0,0,200))

    #draw the arc
    color = 255,0,255
    position = 200,150,200,200
    start_angle = math.radians(0)
    end_angle = math.radians(180)
    width = 8
    pygame.draw.arc(screen, color, position, start_angle, end_angle, width)

    pygame.display.update()
```

The Pie Game

The Pie Game is a very simple game that does not have much by way of difficulty, but it does have a rudimentary level of gameplay and a minor "goodie" when the player "wins." The gameplay involves just pressing the number keys 1, 2, 3, and 4, in any order. As each number is pressed, the corresponding pie piece is drawn. When all four pie pieces are completed, then the pie changes color. The game is shown in Figure 2.7.

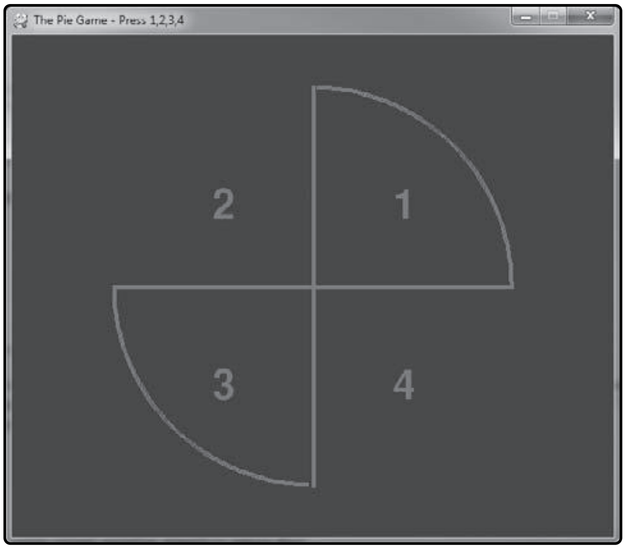


Figure 2.7

The Pie Game with two pieces drawn.

When the player finishes the entire pie, then the color changes to bright green and the numbers and pie shapes are drawn in bright green to reflect that the player has won! This might be a simple game, but it demonstrates a lot of important Pygame concepts that we must learn to become proficient with this library. This game also demonstrates basic logic code in Python, and believe it or not, the very important subject of *state-based programming*. You see, the four pie pieces are not drawn automatically just when the player presses the correct key (1, 2, 3, or 4). Instead, a *state flag* is set when a key is pressed, and that flag is used later to draw the pie pieces based on that flag. This is a *very important* concept, as it demonstrates how to handle events and user interaction *indirectly*.

```
import math
import pygame
from pygame.locals import *
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("The Pie Game - Press 1,2,3,4")
myfont = pygame.font.Font(None, 60)

color = 200, 80, 60
width = 4
x = 300
y = 250
radius = 200
position = x-radius, y-radius, radius*2, radius*2

piece1 = False
piece2 = False
piece3 = False
piece4 = False

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        elif event.type == KEYUP:
            if event.key == pygame.K_ESCAPE:
                sys.exit()
            elif event.key == pygame.K_1:
                piece1 = True
            elif event.key == pygame.K_2:
                piece2 = True
            elif event.key == pygame.K_3:
                piece3 = True
            elif event.key == pygame.K_4:
                piece4 = True

    #clear the screen
    screen.fill((0,0,200))

    #draw the four numbers
    textImg1 = myfont.render("1", True, color)
    screen.blit(textImg1, (x+radius/2-20, y-radius/2))
    textImg2 = myfont.render("2", True, color)
    screen.blit(textImg2, (x-radius/2, y-radius/2))
    textImg3 = myfont.render("3", True, color)
    screen.blit(textImg3, (x-radius/2, y+radius/2-20))
    textImg4 = myfont.render("4", True, color)
    screen.blit(textImg4, (x+radius/2-20, y+radius/2-20))

    #should the pieces be drawn?
    if piece1:
        start_angle = math.radians(0)
        end_angle = math.radians(90)
        pygame.draw.arc(screen, color, position, start_angle, end_angle, width)
        pygame.draw.line(screen, color, (x,y), (x,y-radius), width)
        pygame.draw.line(screen, color, (x,y), (x+radius,y), width)
    if piece2:
        start_angle = math.radians(90)
        end_angle = math.radians(180)
        pygame.draw.arc(screen, color, position, start_angle, end_angle, width)
        pygame.draw.line(screen, color, (x,y), (x,y-radius), width)
        pygame.draw.line(screen, color, (x,y), (x-radius,y), width)
    if piece3:
        start_angle = math.radians(180)
        end_angle = math.radians(270)
        pygame.draw.arc(screen, color, position, start_angle, end_angle, width)
        pygame.draw.line(screen, color, (x,y), (x-radius,y), width)
        pygame.draw.line(screen, color, (x,y), (x,y+radius), width)
    if piece4:
        start_angle = math.radians(270)
        end_angle = math.radians(360)
        pygame.draw.arc(screen, color, position, start_angle, end_angle, width)
        pygame.draw.line(screen, color, (x,y), (x,y+radius), width)
        pygame.draw.line(screen, color, (x,y), (x+radius,y), width)

    #is the pie finished?
    if piece1 and piece2 and piece3 and piece4:
        color = 0,255,0

    pygame.display.update()
```

Summary

This chapter introduced the Pygame library, which will really make our exploration of Python a lot more fun than otherwise plain text output to the console would have been!

Challenges

1. **Using the examples in this chapter as your starting point, write a program that draws an ellipse—one of the shapes we did not cover in the chapter.**
2. **Take one of the examples, such as the line drawing demo, and modify it so that 1,000 lines are drawn with random values. Look at the random library and the `random.randint()` function.**
3. **The Drawing Rectangles Demo is the only one that moved the shape around on the screen. Modify the program so that any time the box hits an edge of the screen it will also change color!**

CHAPTER 3

File I/O, Data, and Fonts: The Trivia Game

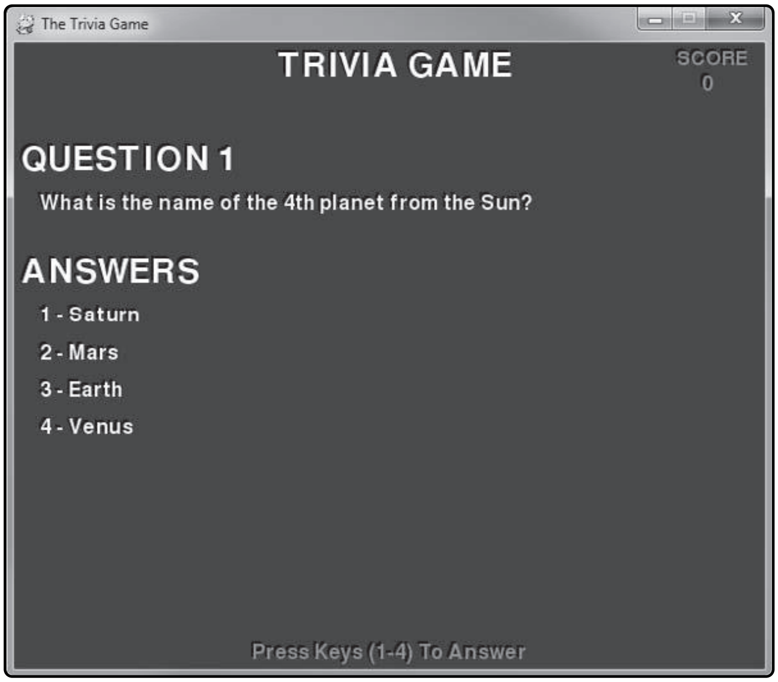
The purpose of a file is to store data in a logical way so that it can be read back later and updated if necessary. To read and write files, then, one has to understand data types, because the data stored in a file must be specific. This chapter explores data types and file input/output. This chapter has a secondary purpose that works well with data types and file I/O: printing text on the screen with fonts.

Here are the topics covered in this chapter:

- Python data types
- Getting user input
- Handling exceptions
- The Mad Lib Game
- Working with text files
- Working with binary files
- The Trivia Game

Examining The Trivia Game

The Trivia Game demonstrates the concepts covered in this chapter by reading trivia questions out of a file and asking the user to choose from the multiple choice answers. The trivia questions and answers can be easily edited with a text editor or even with IDLE. Figure 3.1 shows what the game will look like when you have finished it in this chapter.


The Trivia Game

TRIVIA GAME

SCORE
0

QUESTION 1

What is the name of the 4th planet from the Sun?

ANSWERS

1 - Saturn
2 - Mars
3 - Earth
4 - Venus

Press Keys (1-4) To Answer

FIGURE 3.1

The Trivia Game.

PYTHON DATA TYPES

Remember that Python is an interpreted script language, not a compiled language like C++, so it is much more forgiving to the programmer with many more tolerances. A Python variable can be a string, and then a number, and then a string again, without complaint. For instance:

```
something = 123
print(something)
something = "ABC"
print(something)
```

That code produces this output:

```
123
ABC
```

HINT

Use the `str()` function when you need to convert a number to a string, and either the `int()` or `float()` function to convert a string with a number in it to a numeric variable.

More Printing

The `print()` function can print more than one variable at a time; just separate each one with a comma. For instance, this:

```
A = 123
B = "ABC"
C = 456
D = "DEF"
print(A,B,C,D)
```

produces this:

```
123 ABC 456 DEF
```

Note that `print()` inserts a space in between each item being printed. You can add a blank line to the text output by `print()` by inserting the `\n` character code in a string variable, such as:

```
name = "John Carpenter\n"
birth = "11/11/2011"
print(name,birth)
```

which outputs:

```
John Carpenter
 11/11/2011
```

Note that `print()` still inserted a blank space after the newline character. The `print()` function has a second optional parameter that can specify the separator character, which can be changed from the default space character. In fact, `print()` has four parameters in total! The third specifies the newline character, and the fourth specifies the output destination for redirection (not commonly used).

```
print("String","Theory", sep='-', end=':')
```

produces this output:

```
String-Theory:
```

There are some values built into Python that we can print out. Let's print out `sys.copyright` to display the copyrights of all the modules used in Python. First, add

```
import sys
```

to the program so the sys module is available. Then, print the value:

```
print(sys.copyright)
```

This prints out:

```
Copyright (c) 2001-2011 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

TRAP

Some modules are not automatically loaded by Python. If you think your code is correct but python.exe is still giving you syntax errors, make sure you are using the import statement with the required modules.

Another interesting value is `sys.platform`, which is a string representing the operating system currently in use. This will be "win32" or "win64" for Windows systems, and will vary according to the other operating systems supported by Python. If you ever want to know what version of Python you're using, you can display it with:

```
print(sys.version)
```

which produces output something like this (it will vary from one system to another):

```
3.2 (r32:88445, Feb 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)]
```

How about printing out the date and time? Here's the code:

```
import datetime
from datetime import datetime, timezone
print(datetime.now())
```

which produces this output:

```
2011-06-14 16:52:00.572000
```

There's quite a bit more to the `datetime` class that allows retrieval of specific `datetime` components (month, day, year, etc.).

Getting User Input

We can get user input from the console using the `input()` function, which returns a string. The simplest use of the function is to just pause output when a program has finished running. An optional parameter displays text before waiting for input. For example:

```
poem = """
Three Rings for the elven kings under the sky,
Seven for the dwarf lords in their halls of stone,
Nine for the mortal men doomed to die,
One for the dark lord on his dark throne.
In the land of mordor where the shadows lie,
One ring to rule them all, One ring to find them,
One ring to bring them all and in the darkness bind them.
 - J.R.R. Tolkien
"""
print(poem)
input("Press Enter to continue...")
```

TRICK

To create a lengthy text string on multiple lines, enclose the lines of text in triple quotes.

This example will wait for the user to press the Enter key before exiting. But, we can also read the input typed in from the `input()` function, rather than just look for the Enter key to be pressed (which returns an empty string, by the way).

```
name = input("Pray tell, what is thy name? ")
print("Fare thee well, Master", name)
```

And the output:

```
Pray tell, what is thy name? Ambivalent Programmer
Fare thee well, Master Ambivalent Programmer
```

Python excels at handling groups of information. To create an array-like list of any data, just set a variable equal to data items in brackets, with each one separated with a comma: mylist | [1,2,3,4,5]. Strings and other data types can also be used.

Handling Exceptions

If you need to have the user type in a number and then use it in a calculation, the text entered can be converted to a numeric variable with the `int()` or `float()` function. But, if the user types in non-numeric digits, then the program will crash! We can't allow that to disrupt the program, because users might type in an invalid input value, and without handling the *likelihood* of an error, the program definitely will crash. For example:

```
Enter a number: ABC
Traceback (most recent call last):
  File "InputDemo.py", line 4, in <module>
    number = float(s)
ValueError: could not convert string to float: 'ABC'
```

We can handle this problem with a `try...except` block, which will trap errors. In the example below, the questionable line occurs inside the `try:` block, and the code in the `except:` block will run if there is an error. In either case, the program continues running.

```
s = input("Enter a number: ")
try:
    number = float(s)
except:
    number = 0
answer = number * number
print(number,"*",number,"=",answer)
```

Here is a sample run. If you enter invalid data, then the output will just reflect a value of 0 because of the error handler.

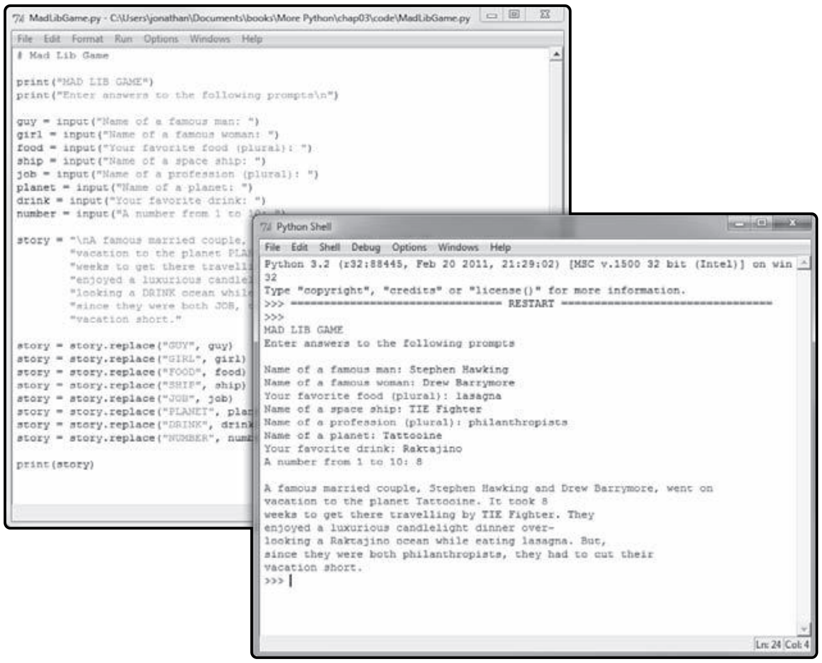
```
Enter a number: 15
15.0 * 15.0 = 225.0
```

The Mad Lib Game

This is not the final chapter example, but we know enough about getting input to make a simple Mad Lib game just for fun. The Mad Lib Game is pretty simple. You ask someone to fill in some names, things, places, and then use those words and phrases to fill in a story, with often humorous results that are unexpected. The interesting thing about this little program is how the story is constructed. Instead of building the story out of the user input variables (guy, girl, food, etc.), it uses `string.replace()` to do a search-and-replace operation on the story string, replacing tagged words (in caps) with user data. There are so many useful classes and methods in the Python modules! Study the Python docs, like an explorer charting an undiscovered country, and learn what great mysteries lie hidden! That is what sets apart a mediocre programmer from a *great* programmer.

```
print("MAD LIB GAME")
print("Enter answers to the following prompts\n")

guy = input("Name of a famous man: ")
girl = input("Name of a famous woman: ")
food = input("Your favorite food (plural): ")
ship = input("Name of a space ship: ")
job = input("Name of a profession (plural): ")
planet = input("Name of a planet: ")
drink = input("Your favorite drink: ")
number = input("A number from 1 to 10: ")

story = "\nA famous married couple, GUY and GIRL, went on\n" +\
        "vacation to the planet PLANET. It took NUMBER\n" +\
        "weeks to get there travelling by SHIP. They\n" +\
        "enjoyed a luxurious candlelight dinner over-\n" +\
        "looking a DRINK ocean while eating FOOD. But,\n" +\
        "since they were both JOB, they had to cut their\n" +\
        "vacation short."

story = story.replace("GUY", guy)
story = story.replace("GIRL", girl)
story = story.replace("FOOD", food)
story = story.replace("SHIP", ship)
story = story.replace("JOB", job)
story = story.replace("PLANET", planet)
story = story.replace("DRINK", drink)
story = story.replace("NUMBER", number)
print(story)
```

Sample output looks like this (note that it will change based on what the user types in). I have highlighted the user data in bold to show how the story was constructed.



```
# Mad Lib Game

print("MAD LIB GAME")
print("Enter answers to the following prompts\n")

guy = input("Name of a famous man: ")
girl = input("Name of a famous woman: ")
food = input("Your favorite food (plural): ")
ship = input("Name of a space ship: ")
job = input("Name of a profession (plural): ")
planet = input("Name of a planet: ")
drink = input("Your favorite drink: ")
number = input("A number from 1 to 10: ")

story = "\nA famous married couple,
        "vacation to the planet PLA
        "weeks to get there travelli
        "enjoyed a luxurious candle
        "looking a DRINK ocean whil
        "since they were both JOB,
        "vacation short."

story = story.replace("GUY", guy)
story = story.replace("GIRL", girl)
story = story.replace("FOOD", food)
story = story.replace("SHIP", ship)
story = story.replace("JOB", job)
story = story.replace("PLANET", pla
story = story.replace("DRINK", drin
story = story.replace("NUMBER", num

print(story)
```

```
Python 3.2 (r32:88445, Feb 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
MAD LIB GAME
Enter answers to the following prompts

Name of a famous man: Stephen Hawking
Name of a famous woman: Drew Barrymore
Your favorite food (plural): lasagna
Name of a space ship: TIE Fighter
Name of a profession (plural): philanthropists
Name of a planet: Tattooine
Your favorite drink: Raktajino
A number from 1 to 10: 8

A famous married couple, Stephen Hawking and Drew Barrymore, went on
vacation to the planet Tattooine. It took 8
weeks to get there travelling by TIE Fighter. They
enjoyed a luxurious candlelight dinner over-
looking a Raktajino ocean while eating lasagna. But,
since they were both philanthropists, they had to cut their
vacation short.
>>>
```

FIGURE 3.2

The Mad Lib Game.

```
MAD LIB GAME
Enter answers to the following prompts

Name of a famous man: **Stephen Hawking**
Name of a famous woman: **Drew Barrymore**
Your favorite food (plural): **lasagna**
Name of a space ship: **TIE Fighter**
Name of a profession (plural): **philanthropists**
Name of a planet: **Tattooine**
Your favorite drink: **Raktajino**
A number from 1 to 10: **8**

A famous married couple, **Stephen Hawking** and **Drew Barrymore**, went on
vacation to the planet **Tattooine**. It took **8**
weeks to get there travelling by **TIE Fighter**. They
enjoyed a luxurious candlelight dinner over
```

```
looking a Raktajino ocean while eating lasagna. But,
since they were both philanthropists, they had to cut their
vacation short.
```

It would take some work to line up the resulting story text evenly on each line. It can be done but the code was kept on the short and simple side for the sake of illustration.

File Input/Output

The simplest form of file is a text file that could be opened with a text editor like Notepad. In such a file, we can read data with one significant item per line and then read each line into a variable.

Working with Text Files

To open a file in Python, use the `open()` function. The first parameter is the filename, and the second is the open mode. The modes are shown in Table 3.1. In most cases, we will just use "r" to read, but all of the modes are available and files can be created, appended, overwritten, and read with the file functions.

Table 3.1 Text File Open Modes

Mode	Description
"r"	Open text file to read data.
"w"	Open text file to write data.
"a"	Open text file to append data.
"r+"	Open text file to read and write data.
"w+"	Open text file to write and read data.
"a+"	Open text file to append and read data.

The `open()` function might be called like so:

```
file = open("data.txt", "r")
```

To close the file after finishing with it:

```
file.close()
```

Writing to a Text File

To write data out to a text file, we have to open the file with the "w" write property. There is one primary way to write text data to a file, using the `file.write()` function. Surprisingly, there is no `writeline()`-type function to write just a singular line (but there is *plural* `file.writelines()` for writing a list of strings), so we have to add a new-line character (`\n`) to the end of text that needs to be saved on a separate line.

```
file = open("data2.txt", "w")
file.write("Sample file writing\n")
file.write("This is line 2\n")
file.close()
```

Here is another example that writes several lines out at once from a string list:

```
text_lines = [
    "Chapter 3\n",
    "Sample text data file\n",
    "This is the third line of text\n",
    "The fourth line looks like this\n",
    "Edit the file with any text editor\n" ]

file = open("data.txt", "w")
file.writelines(text_lines)
file.close()
```

Reading from a Text File

To read from a file, we must first open it for reading. The code is similar to opening a file for writing, but we just need to change the file mode:

```
file = open("data.txt", "r")
```

Once a file has been opened, the data inside can be read, and there are a number of functions available to do this in different ways. To read a single character at a time, use `file.read(n)`, where `n` is the number of characters to read:

```
char = file.read(10)
print(char)
```

reads 10 characters from the file at the current file pointer position. Repeated calls like this will continue to read more characters out of the file and advance the position. To read the whole file into a string variable:

```
all_data = file.read()
print(all_data)
```

We can also read a whole line of text data with `file.readline(n)`, where `n` is an optional number of characters to read from the current line.

```
one_line = file.readline()
print(one_line)
```

To read all of the lines in the entire data file, use `file.readlines()`. Invoking this function does not fill the receiving variable with text data. Instead, a list is created with each line an item in the list. Printing the data in the list variable does not print out the text data as it appears in the file. For instance, this code:

```
all_data = file.readlines()
print(all_data)
```

produces this output:

```
['Chapter 3\n', 'Sample text data file\n', 'This is the third line of text\n',
'The fourth line looks like this\n', 'Edit the file with any text editor\n']
```

Strange looking, wouldn't you agree? Well, since a list was created out of the `all_data` variable, it can be indexed like an array with a `for` loop. Note that the `string.strip()` modifier is used: this removes line-feed characters from the end of lines.

```
print("Lines: ", len(all_data))
for line in all_data:
    print(line.strip())
```

The output shows the contents of the text file:

```
Lines:  5
Chapter 3
Sample text data file
This is the third line of text
The fourth line looks like this
Edit the file with any text editor
```

Working with Binary Files

Binary files contain bytes. The bytes might be encoded integers, encoded floats, encoded lists (written using pickling, which we'll cover here shortly), or any other type of data. A PNG bitmap file can be read with binary file access in Python. Now, interpreting the data afterward is up to you, the programmer, but Python can read the data and supply it in a buffer for processing. Table 3.2 lists the binary file modes.

Table 3.2 Binary File Open Modes

Mode	Description
"rb"	Open binary file to read data.
"wb"	Open binary file to write data.
"ab"	Open binary file to append data.
"rb+"	Open binary file to read and write data.
"wb+"	Open binary file to write and read data.
"ab+"	Open binary file to append and read data.

Opening a file in binary mode is similar to what we've already seen, and might be called like so:

```
file = open("data.txt", "rb")
```

To close the file after finishing with it:

```
file.close()
```

Writing to a Binary File

This is debatable, but I think the most useful sort of binary file is one that contains data corresponding to a Python structure. Our ability to write a structure to a file and read it back with the fields intact will really handle any custom data file needs that we are likely to have, either for a game or any other program. Python has no direct correlation between a user-defined data type structure and file input/output. But it *does* provide a library module called *struct* with the ability to pack data into a *string* for output. We can write this data in binary mode, but interestingly enough, it was really designed for writing text data as a buffer.

The way data is encoded into binary format is with the `struct.pack()` function. When reading the data from the file again, it is decoded with `struct.unpack()`. `struct` is a Python module. To use it, we must include it first with an `import` statement, just like we do with Pygame:

```
import struct
```

Let's see how to read and write a file in binary mode. The following example code writes 1000 integers to a binary file. First, let's see how to write, and then we'll read the data back. First, open the file for binary write mode:

```
file = open("binary.dat", "wb")
```

Next, write out 1000 integers to the file:

```
for n in range(1000):
    data = struct.pack('i', n)
    file.write(data)
```

Lastly, close the file:

```
file.close()
```

Reading from a Binary File

Now we'll see how to read data back out of a binary file and unpack it for display, one value at a time. To verify that the code is working, we should expect to see the values 0 to 999 come up. First, we open the file, and calculate the size of an int with `struct.calcsize()`, so the `struct.unpack()` function will know how many bytes to read for each number.

```
file = open("binary.dat", "rb")
size = struct.calcsize("i")
```

Next, a `while` loop reads the data in the file *size* bytes at a time until all data has been read. As each value is read, it is unpacked, converted from a list to a simple variable, and printed out.

```
bytes_read = file.read(size)
while bytes_read:
    value = struct.unpack("i", bytes_read)
    value = value[0]
    print(value, end=" ")
    bytes_read = file.read(size)
file.close()
```

Similar code could be written to store additional data to the file in sequence. As long as the data is read back out in the same order that it was written, then different types of data can be written to the file.

The Trivia Game

It's time to apply what we've learned about file input/output to a game that will help strengthen your grasp of the subject. The game will run in a graphics window using Pygame, so we will need to learn to use text output.

Printing Text with Pygame

We have been printing a lot of text out to the console in this chapter while learning about file input/output, but there comes a point where the console is no longer sufficient and we need a higher level of user interaction that only a graphical system can provide. We'll bump it up a notch by learning to print text to the screen in graphics mode using Pygame.

The `pygame.font` module gives us the ability to print font-based text to the screen in graphics mode. We've already used `pygame.font` in the previous chapter but we'll review it quickly again. The class that produces a printable font is `pygame.font.Font`. By default, passing `None` as the font name will cause the `pygame.font.Font()` constructor to load the default Pygame font. The second parameter to the constructor is the font point size. This line creates a default font with a 30-point size:

```
myfont = pygame.font.Font(None, 30)
```

We can also specify a font name to choose a custom font for our game:

```
myfont = pygame.font.Font("Arial", 30)
```

To print text, the `font.render()` function creates a bitmap with the text written on it, which we then draw to the screen using `screen.blit()`.

```
image = font.render(text, True, (255,255,255))
screen.blit(image, (100, 100))
```

The Trivia Class

The main program source code in the game is primarily responsible for getting keyboard input and refreshing the screen. The bulk of the gameplay code is found in a new class called `Trivia`. First, we'll import the modules needed for the game:

```
import sys, pygame
from pygame.locals import *
```

Next, we'll get the `Trivia` class started. The constructor, `__init__()`, has a filename parameter that you pass to it, which contains the trivia data. The data is loaded with a single `file.readlines()` function call, and then that data is used in its list by the game. There are quite a few field variables (also called properties) in the `Trivia` class, to handle the game logic. All of the logic is performed by methods in the `Trivia` class, not in the main program.

```
class Trivia(object):
    def __init__(self, filename):
        self.data = []
        self.current = 0
        self.total = 0
        self.correct = 0
        self.score = 0
        self.scored = False
        self.failed = False
        self.wronganswer = 0
        self.colors = [white,white,white,white]
```

Loading the Trivia Data

After the data is loaded, then the trivia data is parsed (from its list object called `trivia_data`) and copied one line at a time into a new list called `Trivia.data`. The reason for the new list is so we can strip each line of whitespace (main line-feed characters at the end of each line). The following code is also found in the constructor, `__init__()`.

```
        #read trivia data from file
        f = open(filename, "r")
        trivia_data = f.readlines()
        f.close()

        #count and clean up trivia data
        for text_line in trivia_data:
            self.data.append(text_line.strip())
            self.total += 1
```

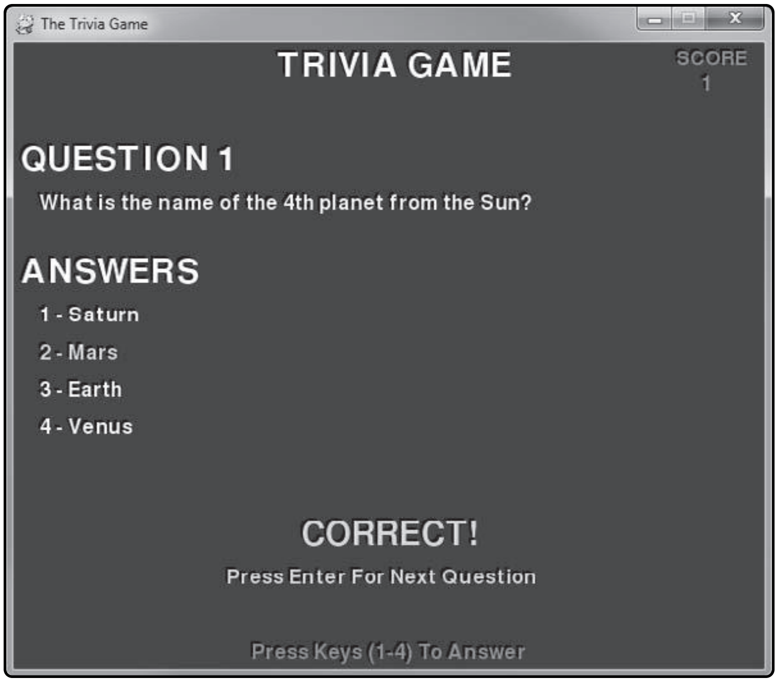
The trivia data file included with the game has only five questions, but the game supports more, so you are welcome to add more questions. The theme of this trivia game is astronomy. Don't cheat and look at the answers before at least trying to answer them on your own! You can edit the trivia_data.txt file with any text editor, including IDLE. The format of the trivia data goes like this: line 1 is the question; lines 2–5 are the answers; line 6 is the correct answer. See, simple!

```
What is the name of the 4th planet from the Sun?
Saturn
Mars
Earth
Venus
2
Which planet has the most moons in the solar system?
Uranus
```

```
Saturn
Neptune
Jupiter
4
Approximately how large is the Sun's diameter (width)?
65 thousand miles
45 million miles
1 million miles
825 thousand miles
3
How far is the Earth from the Sun in its orbit (on average)?
13 million miles
93 million miles
250 thousand miles
800 thousand miles
2
What causes the Earth's oceans to have tides?
The Moon
The Sun
Earth's molten core
Oxygen
1
```

Displaying the Question and Answers

The bulk of the work in the game is found in the `Trivia.show_question()` method. It draws the entire screen for the game: the title, the footer, the score, the question, answers, and does the colorizing of the answers based on user input. When the player chooses the correct answer, it is printed in green. But if they choose the wrong answer, it will be printed in red, and the correct one printed in green. This could have required quite a bit of logic code, but it was simplified using a list of four colors that is used when drawing the text of each answer. The key to indexing from one question record (in the loaded data) to another is the `Trivia.current` field. Figure 3.3 shows the resulting display when the user gets an answer right.



TRIVIA GAME

SCORE
1

QUESTION 1

What is the name of the 4th planet from the Sun?

ANSWERS

1 - Saturn

2 - Mars

3 - Earth

4 - Venus

CORRECT!

Press Enter For Next Question

Press Keys (1-4) To Answer

Figure 3.3 Getting the answer right.

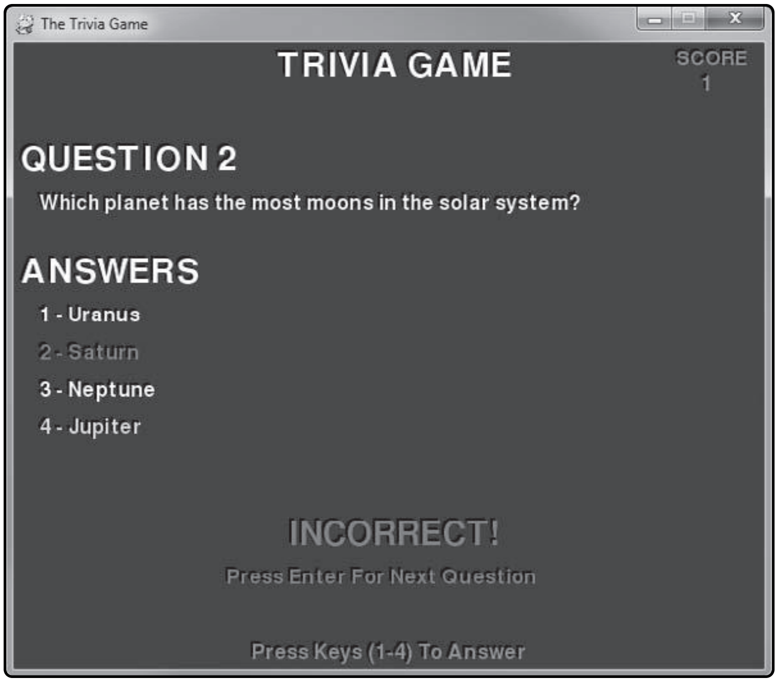
```
def show_question(self):
    print_text(font1, 210, 5, "TRIVIA GAME")
    print_text(font2, 190, 500-20, "Press Keys (1-4) To Answer", purple)
    print_text(font2, 530, 5, "SCORE", purple)
    print_text(font2, 550, 25, str(self.score), purple)

    #get correct answer out of data (first)
    self.correct = int(self.data[self.current+5])

    #display question
    question = self.current // 6 + 1
    print_text(font1, 5, 80, "QUESTION " + str(question))
    print_text(font2, 20, 120, self.data[self.current], yellow)

    #respond to correct answer
    if self.scored:
        self.colors = [white,white,white,white]
        self.colors[self.correct-1] = green
```

```
            print_text(font1, 230, 380, "CORRECT!", green)
            print_text(font2, 170, 420, "Press Enter For Next Question", green)
        elif self.failed:
            self.colors = [white,white,white,white]
            self.colors[self.wronganswer-1] = red
            self.colors[self.correct-1] = green
            print_text(font1, 220, 380, "INCORRECT!", red)
            print_text(font2, 170, 420, "Press Enter For Next Question", red)

        #display answers
        print_text(font1, 5, 170, "ANSWERS")
        print_text(font2, 20, 210, "1 - " + self.data[self.current+1], self.colors[0])
        print_text(font2, 20, 240, "2 - " + self.data[self.current+2], self.colors[1])
        print_text(font2, 20, 270, "3 - " + self.data[self.current+3], self.colors[2])
        print_text(font2, 20, 300, "4 - " + self.data[self.current+4], self.colors[3])
```

Responding to User Input

The Trivia Game works by waiting for the user to press the keys 1, 2, 3, or 4, to choose one of the four answers. When the user presses one of these keys, the `Trivia.handle_input()` method is called. If an answer has not already been chosen, then the user input will be compared to the correct answer, and either `self.scored` or `self.failed` will be set to `True`. The game then responds to these two flags, and is put into a wait state until the user presses the Enter key to continue to the next question.

```
    def handle_input(self,number):
        if not self.scored and not self.failed:
            if number == self.correct:
                self.scored = True
                self.score += 1
            else:
                self.failed = True
                self.wronganswer = number
```



TRIVIA GAME

SCORE
1

QUESTION 2

Which planet has the most moons in the solar system?

ANSWERS

1 - Uranus
2 - Saturn
3 - Neptune
4 - Jupiter

INCORRECT!

Press Enter For Next Question

Press Keys (1-4) To Answer

Figure 3.4

Getting the answer wrong.

Going to the Next Question

After an answer has been chosen, the game displays the result and waits for the user to press the Enter key to continue. That key triggers a call to the method `Trivia.next_question()`. If the game is in the wait state between questions, then the flags are reset, the colors are reset, and the game jumps to the next question. Since there are 6 lines per question in the data file (1 question, 4 answers, and 1 number representing the correct answer), the `Trivia.current` field is incremented by 6 to jump to the next question.

```
def next_question(self):
    if self.scored or self.failed:
        self.scored = False
        self.failed = False
        self.correct = 0
        self.colors = [white,white,white,white]
        self.current += 6
        if self.current >= self.total:
            self.current = 0
```

Main Code

The main code is rather tight since so much gameplay functionality was put into the `Trivia` class. First up, we have a helper function called `print_text()`. It's reusable because the first parameter you should pass to the function is a font object.

```
def print_text(font, x, y, text, color=(255,255,255), shadow=True):
    if shadow:
        imgText = font.render(text, True, (0,0,0))
        screen.blit(imgText, (x-2,y-2))
    imgText = font.render(text, True, color)
    screen.blit(imgText, (x,y))
```

Next, we have the main program initialization code that creates the Pygame window and gets things set up for the game.

```
#main program begins
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("The Trivia Game")
font1 = pygame.font.Font(None, 40)
font2 = pygame.font.Font(None, 24)
white = 255,255,255
cyan = 0,255,255
yellow = 255,255,0
purple = 255,0,255
green = 0,255,0
red = 255,0,0
```

Next, the `trivia` object is created (using the `Trivia` class) and a data file called `trivia_data.txt` is loaded. We'll look at the file in a minute.

```
#load the trivia data file
trivia = Trivia("trivia_data.txt")
```

A `while` loop keeps the game running; it may be considered the *game loop*. Most of the code is involved in getting user input with keyboard events. Then it just clears the screen and calls `trivia.show_question()` to update the current state of the game. The last line updates the screen.

```
#repeating loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        elif event.type == KEYUP:
            if event.key == pygame.K_ESCAPE:
                sys.exit()
            elif event.key == pygame.K_1:
                trivia.handle_input(1)
            elif event.key == pygame.K_2:
                trivia.handle_input(2)
            elif event.key == pygame.K_3:
                trivia.handle_input(3)
            elif event.key == pygame.K_4:
                trivia.handle_input(4)
            elif event.key == pygame.K_RETURN:
                trivia.next_question()

    #clear the screen
    screen.fill((0,0,200))

    #display trivia data
    trivia.show_question()

    #update the display
    pygame.display.update()
```

The Real World start

Python is used in many large-scale software engineering projects due to its versatility. For example, NASA used Python in the development of software for the Space Shuttle program! See the success story here: http://www.python.org/about/success/usa.

Summary

This chapter offered a fairly robust coverage of data types, input and printing, file input/output, and managing data effectively for a game. The end result in The Trivia Game demonstrated how easily Python can handle data of different types very easily.

Challenges

1. **Modify The Mad Lib Game by extending the story with your own scenario using the existing code, with your own user input questions.**
2. **Modify the `trivia_data.txt` data file containing questions for The Trivia Game, adding several new astronomy questions. As an alternative, create your own new trivia questions on any other subject of your choice.**
3. **Modify The Trivia Game so that when the last question has been answered, rather than restarting from the beginning, the game prompts the user whether they would like to play again or just quit.**

CHAPTER 4

User Input: The Bomb Catcher Game

We have only scratched the surface of Python and Pygame up to this point, learning how to print text with different fonts and draw lines and shapes in different colors. Don't get me wrong, there's a lot you can do with just these basic capabilities, but Pygame has so much more to offer! We're going to devote this chapter solely to user input. That is, getting user input with the keyboard and mouse. I have mentioned before that Python takes some getting used to, and Pygame shares that distinction as well because of the nature of Python. Pygame was actually based on another library entirely: SDL. Simple DirectMedia Layer (www.libsdl.org) is an open source library that makes 2D graphics drawing and user input very easy to support on multiple platforms. Since Pygame is based on SDL, most of the SDL features are supported in Pygame. We're going to learn to use the user input features in this chapter while making a real-time game.

Here are the topics covered in this chapter:

- Learning to use Pygame events
- Learning about real-time loops
- Learning about keyboard and mouse events
- Learning to poll the keyboard and mouse device states
- Writing The Bomb Catcher Game

Examining The Bomb Catcher Game

The Bomb Catcher Game is shown in Figure 4.1. This game will help reinforce the information learned during the chapter about player input. Specifically, this game uses the mouse to move a red "paddle" at the bottom of the screen, in order to catch yellow "bombs" falling from the top of the screen.

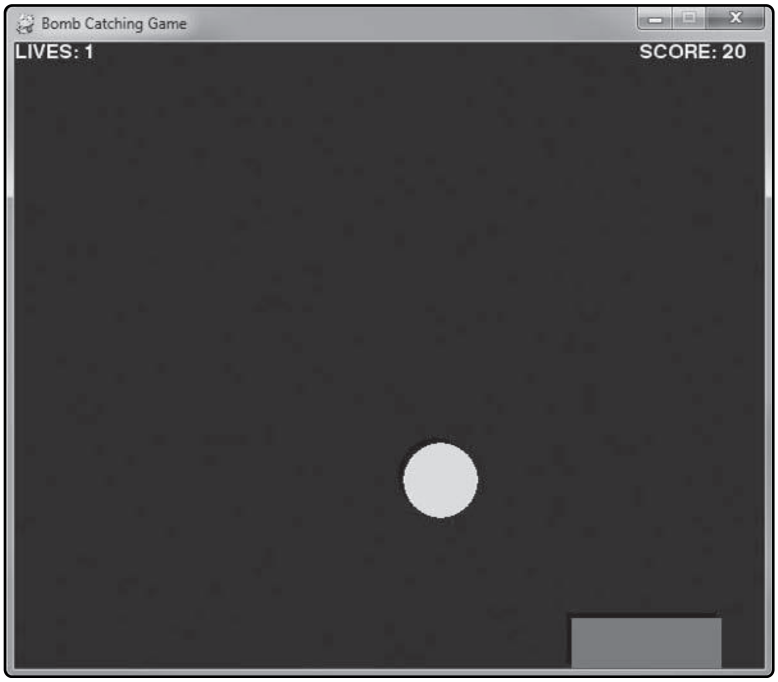


Figure 4.1
The Bomb Catcher Game.

Pygame Events

Pygame events handle a variety of things in a Pygame program. We have already used some of the event types supported by Pygame so they might look familiar to you. Here is the complete list, with the events we've already used in bold:

- **QUIT**
- ACTIVEEVENT
- **KEYDOWN**
- **KEYUP**
- **MOUSEMOTION**
- **MOUSEBUTTONUP**

- `MOUSEBUTTONDOWN`
- `JOYAXISMOTION`
- `JOYBALLMOTION`
- `JOYHATMOTION`
- `JOYBUTTONUP`
- `JOYBUTTONDOWN`
- `VIDEORESIZE`
- `VIDEOEXPOSE`
- `USEREVENT`

We are not going to address all of the event types, just those related to user input. It is true that Pygame supports joystick input. The joystick must be plugged in and configured with the operating system in order for it to work with Pygame. If you wish to try your hand at joystick programming, by all means, go for it! The code will be similar to what we're looking at here for the keyboard and mouse.

TRICK

It is possible to use both the event system and polling to get keyboard and mouse input with Pygame. A combination of the two or just use of one or the other will be based on preference, as it works either way.

Real-Time Event Loop

Event handling in Pygame is done in a real-time loop. A loop is created with the `while` statement and a `while` block, in which all code inside the `while` block is executed repeatedly as long as the `while` condition remains true. In many of the examples shown in this book, we have used

```
while True:
```

as the conditional qualifier. This code normally would create an infinite loop, and it does, except that we have an out with the `sys.exit()` function.

To respond to a Pygame event, we have to parse the events and look at each one. While there can be many events happening at a time, there will usually be just one type of event in a simple demo. More complex programs, and especially games, will have many events happening at the same time. So, we need to parse the events as they are generated. This is done using

```
pygame.event.get()
```

This code will create a list of events that are currently waiting to be processed. We go through the complete list with a `for` loop:

```
for event in pygame.event.get():
```

which gives us each event in the queue as they are generated. Typical events will be key presses, key releases, and mouse movement. The most common event to which we must respond is `QUIT`, which happens when the window is closed by the user.

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
```

Keyboard Events

The keyboard events include `KEYUP` and `KEYDOWN`. When you want to respond to a key being pressed, look at the `KEYDOWN` event, and for released, `KEYUP`. Often, the best way to respond to key events is with flag variables. For instance, when the Space key is pressed, a flag such as `space_key = True` is set. Then, when the key is released, `space_key = False` is set. In this way, we don't have to respond to events immediately as they happen, but can respond to the flag variable instead (elsewhere in the program).

One common key to look for in nearly every program is the quit key. Normally, I use Escape as the default quit key, as a standard way to end a program. We can code a response to the Escape key like so:

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        elif event.type == KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                sys.exit()
```

Note in this example that an `elif` statement had to be used when evaluating the event types. There is no "switch" or "select" conditional statement in Python, only `if...elif...else`. This works pretty well when we only need to work with a few keys. What if we want to look for input from a *large* number of keys? In that case, do we have to write an `if` statement for *every key*? One way is to look at the `key.name` property, which will return a string containing the key name. Another way is to poll the keyboard (more on this later).

By default, Pygame does not respond repeatedly to a key that is being held down; it only sends an event the first time the key is pressed, and then another when it is released. To cause

Pygame to generate repeat events as a key is being held down, we have to turn on the key repeat:

```
pygame.key.set_repeat(10)
```

The parameter is a millisecond repeat value. Calling this method without the parameter disables the key repeat feature.

Mouse Events

The mouse events supported by Pygame include these: `MOUSEMOTION`, `MOUSEBUTTONUP`, and `MOUSEBUTTONDOWN`. The Pygame documentation is a bit sparse regarding the properties of each event, so it takes a little digging to find them. We can read these properties in the event handler when the appropriate event comes up.

For the `MOUSEMOTION` event, the properties are `event.pos`, `event.rel`, and `event.buttons`.

```
for event in pygame.event.get():
    if event.type == MOUSEMOTION:
        mouse_x,mouse_y = event.pos
        move_x,move_y = event.rel
```

For both the `MOUSEBUTTONDOWN` and `MOUSEBUTTONUP` events, the properties are `event.pos` and `event.button`.

```
for event in pygame.event.get():
    elif event.type == MOUSEBUTTONDOWN:
        mouse_down = event.button
        mouse_down_x,mouse_down_y = event.pos
    elif event.type == MOUSEBUTTONUP:
        mouse_up = event.button
        mouse_up_x,mouse_up_y = event.pos
```

Device Polling

The event system in Pygame is not the only means at our disposal for detecting user input. We can also *poll* the input devices to see if the user is interacting with our program.

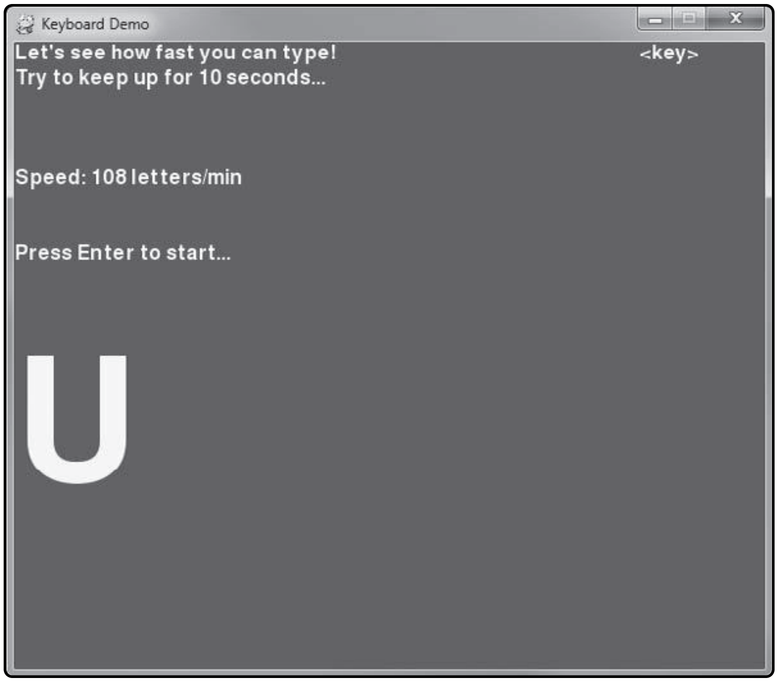
Polling the Keyboard

The interface to keyboard polling in Pygame is with `pygame.key,get_pressed()`. This method returns a list of bools, which is a big list of flags, one per key. The same key constant values are used to index the resulting array of bools (such as `pygame.K_ESCAPE`). The benefit to polling all of the keys at once is the ability to detect multiple key presses without going through the event system. We could replace the old event handler code to detect the Escape key with the following:

```
keys = pygame.key.get_pressed()
if keys[K_ESCAPE]:
    sys.exit()
```

HINT All of the key code constants in Pygame, such as `K_RETURN`, correspond to their ASCII code equivalents, so it is easy to look up a key with any ASCII chart.

We can use a Python function called `chr()` to return the character string representation of an ASCII code number. For instance, the lowercase letter 'a' is ASCII code 97. Here is a short game that uses the keyboard and a real-time loop to test your typing speed! It's not as accurate without whole words—we're only testing typing speed one letter at a time, but it is still a good way to get a handle on the keyboard polling code and support functions. Figure 4.2 shows the program running. Can you beat my score?



```
Let's see how fast you can type!                    <key>
Try to keep up for 10 seconds...


Speed: 108 letters/min


Press Enter to start...



U
```

FIGURE 4.2

The Keyboard Demo tests your typing speed.

```
import sys, random, time, pygame
from pygame.locals import *

def print_text(font, x, y, text, color=(255,255,255)):
    imgText = font.render(text, True, color)
    screen.blit(imgText, (x,y))

#main program begins
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("Keyboard Demo")
font1 = pygame.font.Font(None, 24)
font2 = pygame.font.Font(None, 200)
white = 255,255,255
yellow = 255,255,0

key_flag = False
correct_answer = 97 # "a"
seconds = 11
score = 0
clock_start = 0
game_over = True

#repeating loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        elif event.type == KEYDOWN:
            key_flag = True
        elif event.type == KEYUP:
            key_flag = False

    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]:
        sys.exit()

    if keys[K_RETURN]:
        if game_over:
            game_over = False
            score = 0
            seconds = 11
            clock_start = time.clock()

    current = time.clock() - clock_start
    speed = score * 6
    if seconds-current < 0:
        game_over = True
    elif current <= 10:
        if keys[correct_answer]:
            correct_answer = random.randint(97,122)
            score += 1

    #clear the screen
    screen.fill((0,100,0))

    print_text(font1, 0, 0, "Let's see how fast you can type!")
    print_text(font1, 0, 20, "Try to keep up for 10 seconds...")

    if key_flag:
        print_text(font1, 500, 0, "<key>")

    if not game_over:
        print_text(font1, 0, 80, "Time: " + str(int(seconds-current)))

    print_text(font1, 0, 100, "Speed: " + str(speed) + " letters/min")

    if game_over:
        print_text(font1, 0, 160, "Press Enter to start...")

    print_text(font2, 0, 240, chr(correct_answer-32), yellow)

    #update the display
    pygame.display.update()
```

There's quite a bit of new Python code in this small program that would be helpful to study. Did you notice the random module being used? Look for the function called `random.randint()`. This is a really helpful function that will generate a random number inside the range (supplied by two parameters). Another very helpful new module is `time`, which we also have not seen before. The `time.clock()` function returns the current number of seconds (with milliseconds included as a decimal value) since the program started. I'm using `time.clock()` here in a subtraction calculation to come up with a countdown from 10 down to 1. A variable called `seconds` starts off at 11, and `time.clock()` is subtracted from `seconds` to arrive at a countdown value. Very useful indeed!

Polling the Mouse

We can also ignore the event system and poll the mouse direction if that would work better for our needs. There are really just three mouse functions that we need to learn about. The first one is `pygame.mouse.get_pos()`, which returns the x and y value pair representing the mouse's current position:

```
pos_x,pos_y = pygame.mouse.get_pos()
```

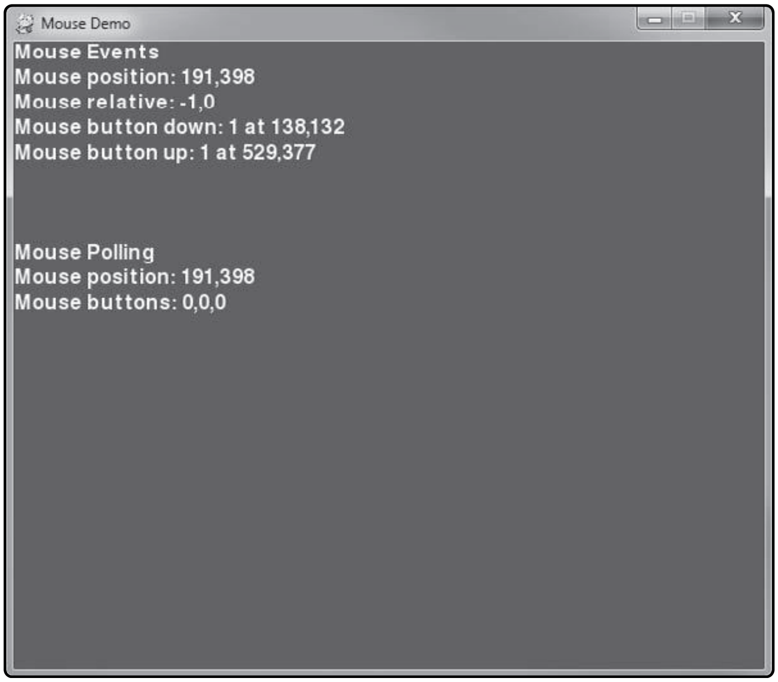
Likewise, we can read the mouse's relative movement in a similar manner with `pygame.mouse.get_rel()`:

```
rel_x,rel_y = pygame.mouse.get_rel()
```

Mouse buttons are read with a call to `pygame.mouse.get_pressed()`, which returns an array of the button states.

```
button1, button2, button3 = pygame.mouse.get_pressed()
```

Below is a complete example of mouse input that demonstrates both event and polled mouse input reading. Figure 4.3 shows the Mouse Demo running.



```
Mouse Events
Mouse position: 191,398
Mouse relative: -1,0
Mouse button down: 1 at 138,132
Mouse button up: 1 at 529,377


Mouse Polling
Mouse position: 191,398
Mouse buttons: 0,0,0
```

Figure 4.3

The Mouse Demo just displays basic mouse status values.

```
import sys, pygame
from pygame.locals import *

def print_text(font, x, y, text, color=(255,255,255)):
    imgText = font.render(text, True, color)
    screen.blit(imgText, (x,y))

#main program begins
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("Mouse Demo")
font1 = pygame.font.Font(None, 24)
white = 255,255,255

mouse_x = mouse_y = 0
move_x = move_y = 0
mouse_down = mouse_up = 0
mouse_down_x = mouse_down_y = 0
mouse_up_x = mouse_up_y = 0

#repeating loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        elif event.type == MOUSEMOTION:
            mouse_x,mouse_y = event.pos
            move_x,move_y = event.rel
        elif event.type == MOUSEBUTTONDOWN:
            mouse_down = event.button
            mouse_down_x,mouse_down_y = event.pos
        elif event.type == MOUSEBUTTONUP:
            mouse_up = event.button
            mouse_up_x,mouse_up_y = event.pos

    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]:
        sys.exit()
    screen.fill((0,100,0))

    print_text(font1, 0, 0, "Mouse Events")
    print_text(font1, 0, 20, "Mouse position: " + str(mouse_x) +
               "," + str(mouse_y))
    print_text(font1, 0, 40, "Mouse relative: " + str(move_x) +
               "," + str(move_y))

    print_text(font1, 0, 60, "Mouse button down: " + str(mouse_down) +
               " at " + str(mouse_down_x) + "," + str(mouse_down_y))

    print_text(font1, 0, 80, "Mouse button up: " + str(mouse_up) +
               " at " + str(mouse_up_x) + "," + str(mouse_up_y))

    print_text(font1, 0, 160, "Mouse Polling")

    x,y = pygame.mouse.get_pos()
    print_text(font1, 0, 180, "Mouse position: " + str(x) + "," + str(y))

    b1, b2, b3 = pygame.mouse.get_pressed()
    print_text(font1, 0, 200, "Mouse buttons: " +
               str(b1) + "," + str(b2) + "," + str(b3))

    pygame.display.update()
```

The Bomb Catcher Game

The final chapter example is called The Bomb Catcher Game. It is actually just a very simple demonstration of mouse input combined with some drawing of basic shapes and a smidgen of collision detection logic. The “bomb” is really just a yellow circle that falls down from the top of the screen over and over again. When the “bomb” reaches the bottom of the screen, the player missed the catch and loses a life (the lives are displayed at the upper left). But if the bomb hits the paddle, then it is caught and another bomb falls. Each time the player catches a bomb, they receive 10 points (score is displayed at the upper right). Figure 4.4 shows the game.

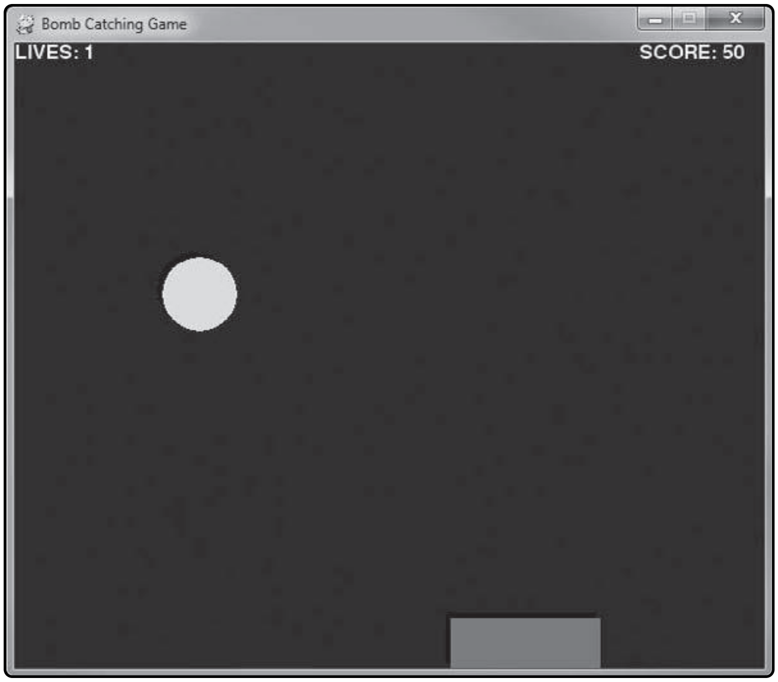


FIGURE 4.4

The Bomb Catching Game.

TRAP Be careful when moving a game object (like our trusty bomb here in this game) using a floating-point velocity value. When converting from a float to an integer, not only is precision lost, but it's possible for game objects to get stuck if they go off the screen due to rounding! I recommend not converting floats to integers except at the point where the integer form is needed (for drawing or printing), but maintain the position of game objects using floats.

```
# Bomb Catcher Game
# Chapter 4
import sys, random, time, pygame
from pygame.locals import *

def print_text(font, x, y, text, color=(255,255,255)):
    imgText = font.render(text, True, color)
    screen.blit(imgText, (x,y))

#main program begins
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("Bomb Catching Game")
```

```
font1 = pygame.font.Font(None, 24)
pygame.mouse.set_visible(False)
white = 255,255,255
red = 220, 50, 50
yellow = 230,230,50
black = 0,0,0

lives = 3
score = 0
game_over = True
mouse_x = mouse_y = 0
pos_x = 300
pos_y = 460
bomb_x = random.randint(0,500)
bomb_y = -50
vel_y = 0.7

#repeating loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        elif event.type == MOUSEMOTION:
            mouse_x,mouse_y = event.pos
            move_x,move_y = event.rel
        elif event.type == MOUSEBUTTONUP:
            if game_over:
                game_over = False
                lives = 3
                score = 0

    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]:
        sys.exit()

    screen.fill((0,0,100))

    if game_over:
        print_text(font1, 100, 200, "<CLICK TO PLAY>")
```

```
else:
    #move the bomb
    bomb_y += vel_y

    #has the player missed the bomb?
    if bomb_y > 500:
        bomb_x = random.randint(0, 500)
        bomb_y = -50
        lives -= 1
        if lives == 0:
            game_over = True

    #see if player has caught the bomb
    elif bomb_y > pos_y:
        if bomb_x > pos_x and bomb_x < pos_x + 120:
            score += 10
            bomb_x = random.randint(0, 500)
            bomb_y = -50

    #draw the bomb
    pygame.draw.circle(screen, black, (bomb_x-4,int(bomb_y)-4), 30, 0)
    pygame.draw.circle(screen, yellow, (bomb_x,int(bomb_y)), 30, 0)

    #set basket position
    pos_x = mouse_x
    if pos_x < 0:
        pos_x = 0
    elif pos_x > 500:
        pos_x = 500
    #draw basket
    pygame.draw.rect(screen, black, (pos_x-4,pos_y-4,120,40), 0)
    pygame.draw.rect(screen, red, (pos_x,pos_y,120,40), 0)

#print # of lives
print_text(font1, 0, 0, "LIVES: " + str(lives))

#print score
print_text(font1, 500, 0, "SCORE: " + str(score))

pygame.display.update()
```

Summary

That's about all we need to know about keyboard and mouse input in order to put them to good use in just about any type of program, including a game. You know, there's a lot we can do with Python and Pygame beyond just games (although that is a fun subject). What about using the code in this chapter to make a drawing program with the ability for the user to save and load their drawings?

Challenges

1. **The Bomb Catching Game is so minimal that it's not very much fun to play. After all, it's just a glorified mouse demo. How about sprucing it up a bit? First, we need a delay when the bomb hits the bottom of the screen. That's supposed to reflect that that bomb "blew up," but nothing happens! When the bomb hits the bottom, make the game pause for a bit, display a "BOOM!" message or something, and wait for the user to click the mouse again before continuing. Or, better yet, use the timing code shown in the Keyboard Demo earlier in the chapter to pause the game for a couple seconds after the bomb goes off, then continue.**
2. **Using the `pygame.draw.arc()` function, add a fuse to the top of the bomb and draw it with a random color repeatedly so the fuse looks like it's actually burning! It may take some work to get the fuse oriented in the right direction. If you need help, turn back to Chapter 2, which covered arc in details (remember The Pie Game?).**
3. **To add some challenge to The Bomb Catching Game, make it so the bomb falls down at an angle by adding a `vel_x` variable and using it (along with the current `vel_y`) to move the bomb! Just be sure to use a small `vel_x` value so the bomb doesn't go off the left or right edge of the screen before it reaches the bottom.**

CHAPTER 5

Math and Graphics: The Analog Clock Demo

This chapter covers the math module of Python that can perform calculations such as the common trigonometry functions sine, cosine, tangent, and others. We will learn to use these and more functions in the math module that are commonly needed for even the simplest of games. To make the math more interesting, we will learn to draw a circle manually and then use that code to create an analog clock with moving hands for the hours, minutes, and seconds. This will also help lead in to the upcoming chapters on bitmaps and sprite animation.

Here are the topics covered in this chapter:

- Learning to use basic trigonometry
- A little circle theory
- Traversing a circle's perimeter
- Drawing a circle manually with sine and cosine
- Creating The Analog Clock Demo

Examining The Analog Clock Demo

The Analog Clock Demo shows how to use some of the math functions in Python (covered in this chapter) to cause the hands of an analog clock to rotate around the clock face. See Figure 5.1.

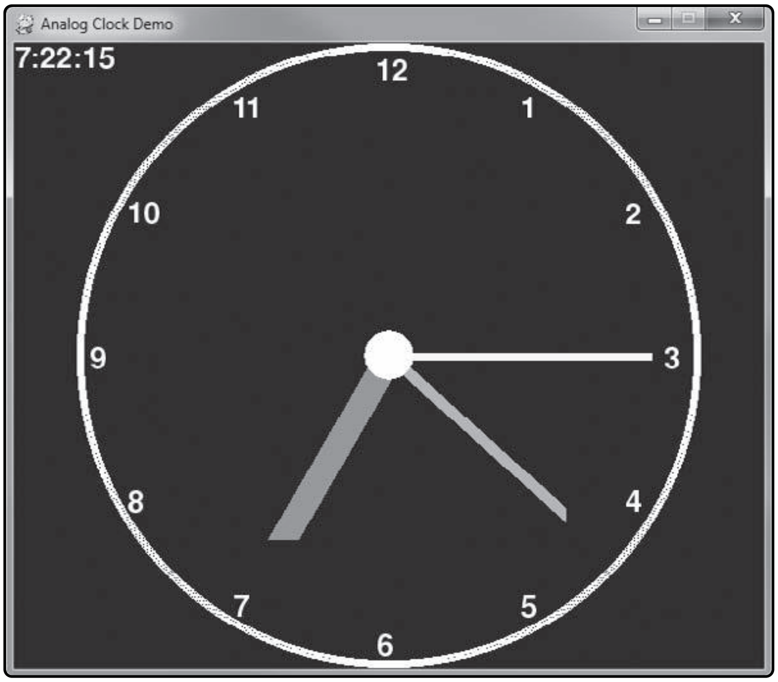


Figure 5.1
The Analog Clock Demo.

Basic Trigonometry

We're just going to learn about a few of the functions in the math module of Python, not all of them. There are several math functions that are consistently used in nearly every video game you are likely to have played—every *serious* game, that is, not just examples or demos. Rotation of a sprite or a mesh (the technical name of a "3D object") is done with two very important math functions: sine and cosine.

If you have had a geometry class, I'm sure you have learned about them. In the old days of video game development, sine and cosine were a problem, because they took a *lot* of CPU cycles to perform one calculation! It was a bit of a performance problem on most PCs, in fact, so an optimization was invented. Take every angle around a circle, 0 to 359, and pre-calculate sine and cosine for every angle, then store the results in an array. Some games would perform this pre-calculation at the start of the game, and it could take several seconds. Some programmers would display a loading screen or introduction to the game while processes like this were running, rather than just make the player wait.

Circle Theory

Working with angles from 0 to 359 involves *degrees*, but the natural "language" of trigonometry functions is *radians*, due to the way a circle is calculated. You might recall that the circumference (or outside perimeter) of a circle can be calculated with this formula:

```
C = [PI] * 2 * Radius (or [PI] * Diameter)
```

where [PI] = 3.14. Python's math module defines [PI] for us with a lot more decimal digits as `math.pi`. Any time you include the math module with `import math` in the program, then `math.pi` can be used in code. It is approximately 3.14159265358979. Sure, you could just use that number in the program yourself and get similar results. Figure 5.2 illustrates the calculation.

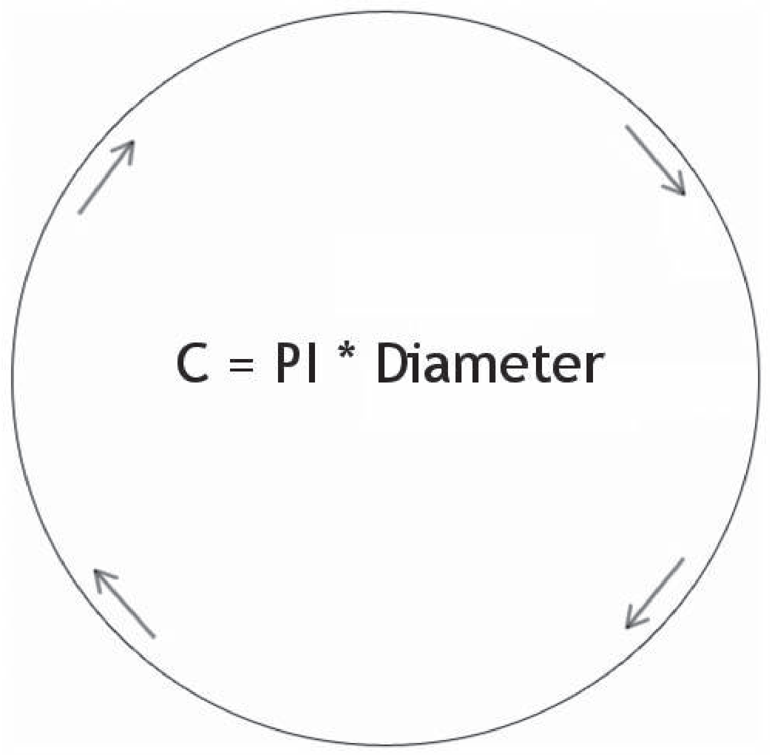


FIGURE 5.2

Calculating the circumference of a circle.

At a certain point, the number of decimal digits only increase precision by a tiny amount. Now, if we're talking about a NASA spacecraft that is travelling billions of miles (like the Voyager craft that flew past Neptune, Pluto, and left the solar system entirely), then you want quite a few digits because in the millions and billions of miles, the decimal precision is a factor! What happens if you walk 1 mile in a certain direction but you are off by just 1 degree? Not a big deal; it might be a matter of a few inches off course. Extend it to 10 miles, 20 miles, 100 miles, and what happens? That 1 degree will put you a long ways off from the desired target. Now imagine the Voyager spacecraft heading out toward Neptune, which is about 2.8 billion miles from the sun. Sometimes such a number is hard for the human mind to grasp, so let's try some conversions: it is equal to twenty-eight hundred *million* miles ! It's like going clear across the United States one million times, or travelling to the Moon and back 6,000 times! Yes, it's that far, which is why precision is so important. In contrast, Mars is only about 30 million miles from Earth at closest approach, and we do not quite have the technology yet today to send a manned mission there.

In The Real World

By the way, the problem of getting to Mars is not so much the distance as it is *solar radiation* that's dangerous to life in deep space. A space ship with its own magnetic field is needed to keep the occupants safe from radiation. If you are a fan of *Star Trek* and similar sci-fi dramas, that was the original purpose for "force fields," aka "shields."

Using the formula, let's calculate the circumference of a sample circle with a diameter of, oh, let's use the Moon's diameter of 2,159 miles (3,474 km).

```
C = [PI] * Diameter
C = 3.14159265358979 * 2159 miles
C = 6,782.6985 miles
```

Do we really need to know the ".6985" part of a mile? That is about 70% of a mile, or 3,688 feet. What if we round it off to just .69? That results in 3,643 feet, an error of 45 feet! That isn't very far considering the large number of miles we're dealing with here, but what if you were a rocket scientist working for NASA on the Apollo missions and had to make sure the ship landed in just the right spot? Now, imagine this compounded by *billions* of miles! Anyway, this is the reason why you want to use as many decimal digits as possible when doing rocket science! Python will use as many digits for [PI] as the computer supports (usually a *double*, which is a C++ data type that supports "*double* precision floating point" numbers with *thousands* of decimal digits!).

We have learned how to *measure* a circle, so now let's learn how to *create* one. A circle can be simulated at any radius size using sine and cosine. The starting point, angle 0, is not at the top like you might naturally assume. The starting point of a circle is at the angle we would think of as 90 degrees from the top toward the right, as shown in Figure 5.3. All circle calculations are based on this angle being the starting point of 0.

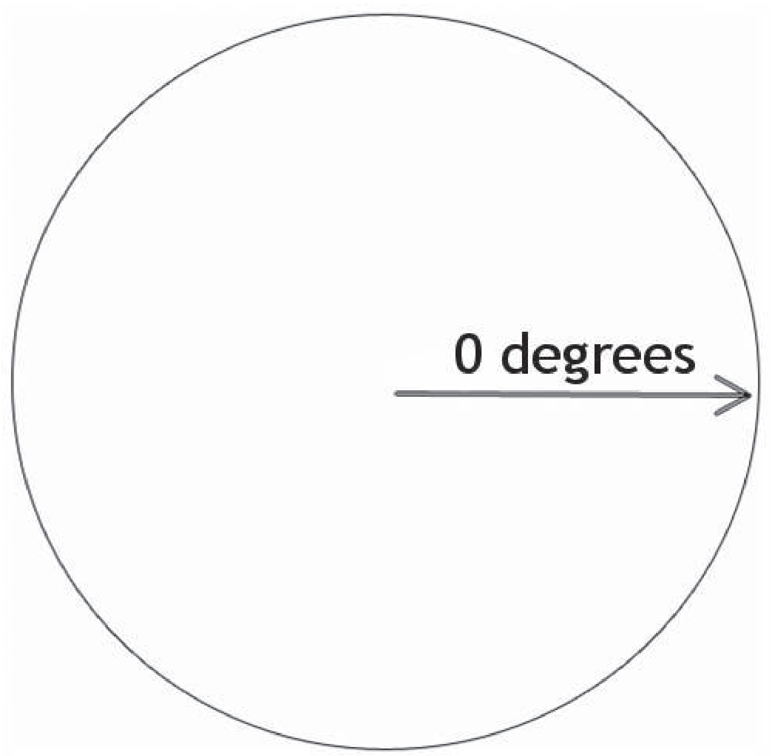


FIGURE 5.3
The starting point of a circle is at the 90-degree point.

Going around the circle from this starting point, a full circle in radians is 2 * [PI] radians, equal to 360 degrees. We can calculate 2 * [PI] approximately as:

```
2 * 3.14159265358979 = 6.28318530715978
```

Are these digits making your head spin? Don't sweat it, just round off at any point you want! 6.28 is perfectly acceptable for our purposes. So, a full circle is 6.28 radians. We can use this to calculate the number of degrees in one radian:

```
360 / 6.28 = 57.3248
```

Likewise, we can calculate the number of *radians* in one *degree*:

```
6.28 / 360 = 0.0174
```

You could use these numbers to convert between degrees and radians with acceptable precision for most video games. Figure 5.4 shows a circle with four cardinal positions labeled. This is also important in a video game, because in most cases we need to wrap the angle around at the 360- degree (2*[PI] radian) point when doing rotations or revolutions.

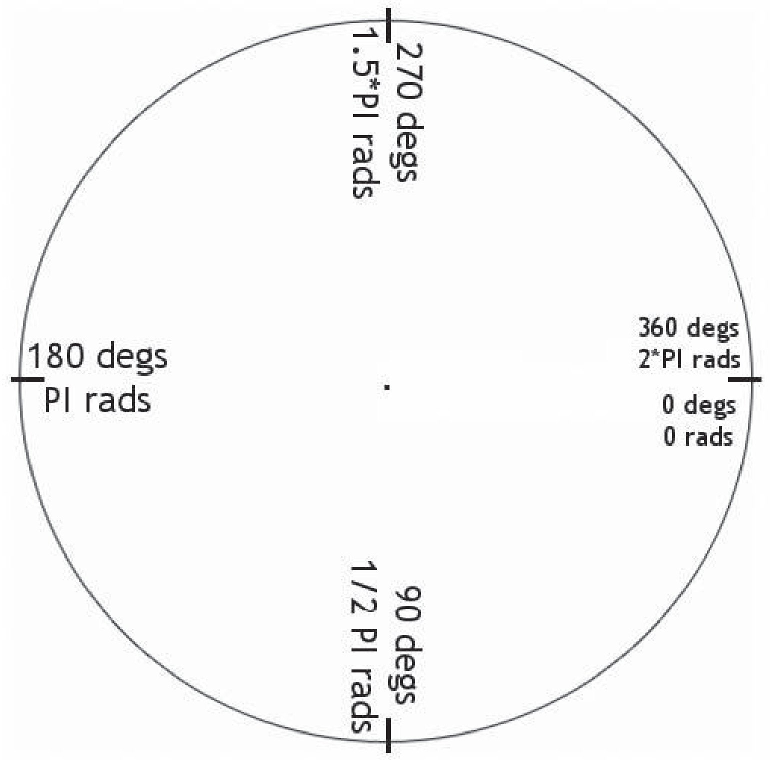


FIGURE 5.4
Degrees and radians around a circle.

Now that you know how to do it the hard way, will you hate me if I tell you it's built in to Python already? You can use `math.degrees()` and `math.radians()` to convert between them!

HINT

If you want to look up the complete reference that lists all of the math functions, go to the reference site at http://docs.python.org/py3k/library/math.html.

Traversing a Circle's Perimeter

Are you enjoying this introduction to rocket science? I hope so, because it gets better! We can sort of "walk" around the perimeter of a circle using the trigonometry functions sine and cosine. All we need to know is the angle and radius. This has huge ramifications in most video games. This algorithm we're about to learn is used in RTS (real-time strategy) games to make units move to the point on the map where you want them to go! This algorithm is also used in just about any shooting game to calculate the direction of a bullet or missile or laser beam. To calculate a point on the circle, we have to get the X and Y value for the coordinate. What's a coordinate? It's a point on the Cartesian coordinate system, shown in Figure 5.5. The X axis right is positive, the Y axis up is positive, and this represents the computer screen, with the origin (0,0) at the upper-left corner of the screen. The other three quadrants still exist! They are just outside the boundary of the screen! It's just interesting to note that it's easy to take for granted that all of the technology we use today someone had to figure out by trial and error, and once figured out, it evolved quickly.

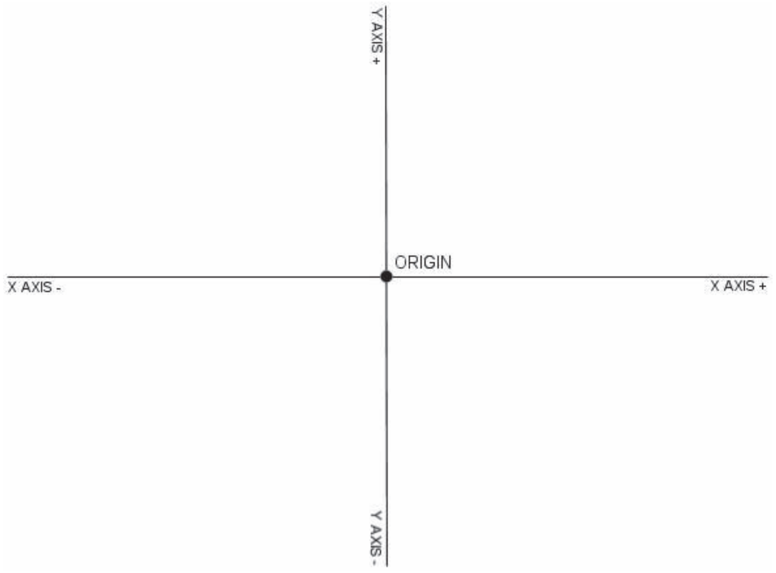


FIGURE 5.5

X and Y axes on a Cartesian coordinate system.

Calculating X

To calculate the X coordinate of any point around the perimeter of a circle, use the cosine function. In Python, this is called `math.cos()`. These functions all require radians as a parameter, not *degrees*. So, if we have to supply radians, but you prefer to work with degrees in your code, then just convert degrees to radians on the fly. For instance:

```
X = math.cos( math.radians(90) )
```

This will be a very small number. All of them will be, at any point around the circle! To see the answer quickly, open up a Python shell and type this:

```
>>> import math
>>> math.cos(math.radians(90))
6.123233995736766e-17
```

That isn't a very easy-to-read number because it's in scientific notation. It has to be because this is a *very tiny* number! There are 16 zeroes after the decimal point before we start seeing the value. To see a number formatted in a way that's easier to read, we have to format the output. Create a string with the formatting codes in it, and that string becomes a string class, which has a `format()` function. Pass your decimal variable to `string.format()` as a parameter. This is one of the cases where Python's great versatility makes it also incredibly confusing at first. Here's an example:

```
>>> '{:.2f}'.format(X)
'0.00'
```

Oh no, there's nothing there! This is not an error; it just means the number is much, much smaller than two digits can show. In fact, this number is not just small, it's infinitesimally small. The rather granular values typically used in a video game would never use such a small number. So, for all practical purposes, this number *is* zero. What we're seeing is just a very tiny remnant of the cosine calculation. Let's get it out to 20 digits:

```
>>> '{:.20f}'.format(X)
'0.00000000000000006123'
```

There, we can start to see the number rounded to 20 digits. Let's extend it out some more to see what happens:

```
>>> '{:.30f}'.format(X)
'0.000000000000000061232339957368'
>>> '{:.40f}'.format(X)
'0.0000000000000000612323399573676603586882'
>>> '{:.50f}'.format(X)
'0.00000000000000006123233995736766035868820147291983'
>>> '{:.60f}'.format(X)
'0.000000000000000061232339957367660358688201472919830231284606'
```

60 digits is getting kind of ridiculous, but it's helpful to see how Python stores the number in our X variable with such high precision. Note that the first 16 digits after the decimal are zero—those are the only important digits. The number really is zero. As we go around the perimeter of the circle, from 0 to 359 degrees, we will see small values, but *nothing* this small. We don't need to print this number out anyway, as it turns out, we just need to use it to calculate the boundary of the circle at 90 degrees (which is due south—remember the orientation!). The next step is to multiply this value by the radius. Think of it as a microscopic circle around the origin point that's so small it looks like nothing more than a single point.

HINT

The Python 3.2 online reference manual for string formatting is found at http://docs.python.org/py3k/library/string.html#formatspec. There are many more ways to format numbers than the one method shown here to represent decimal numbers.

Calculating Y

Let's calculate the Y part of the coordinate so we can begin traversing the perimeter of a circle. To calculate Y, we use sine, which in Python is done with `math.sin(angle)`. Let's have the Python shell prompt do it for us:

```
>>> math.sin( math.radians(90) )
1.0
```

Would you look at that! We have a *normal* number of 1.0! Well, let's think about this for a minute. 90 degrees on the circle is due south (remember, the starting point is to the right, due east on the circle). When we want to represent a point due south from any given location, what is one way to represent that?

```
( X = 0.0, Y = 1.0 )
```

Multiplying the resulting value by a radius factor causes the point to be moved from the origin (at the center of the circle) out to the perimeter. So, together, we have this new algorithm for calculating each point on the perimeter around a circle:

```
X = math.cos( math.radians( angle ) ) * radius
Y = math.sin( math.radians( angle ) ) * radius
```

Circle Demo

I think that's more than enough on circle theory! Actually, would you believe there's more yet? It's true, there's more on this subject, but it's all incredibly valuable information because these concepts are at the core of most video games. Everything from making a space ship move in a certain direction, to causing a tank to fire its cannon, to having pool cue balls bounce off each other realistically can be done with these concepts related to circle theory. Let's get an example up and running so we can see some code in action. Figure 5.6 shows the Circle Demo program running. Every time the angle reaches 360, a new random color is chosen and the circle is drawn again, one degree at a time.

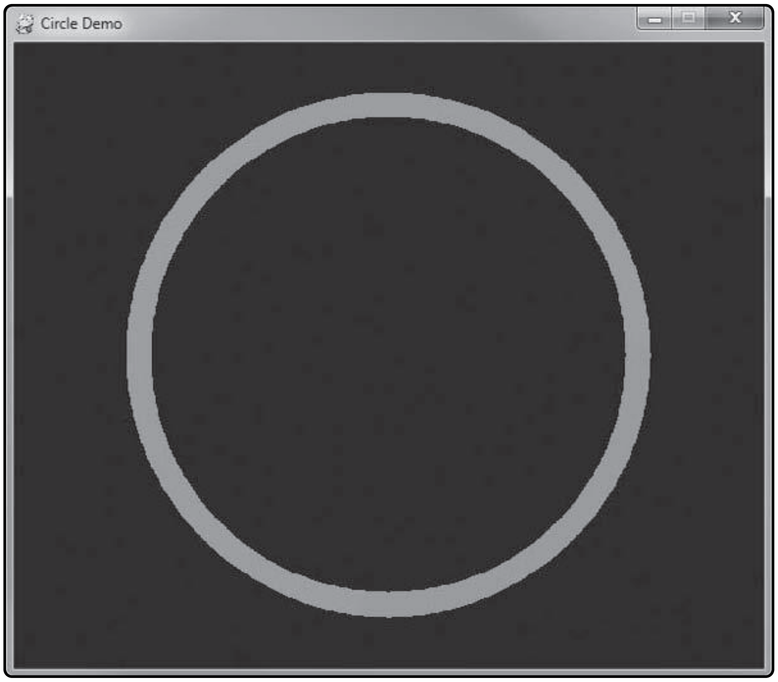


Figure 5.6

Drawing a circle the "hard way."

```
import sys, random, math, pygame
from pygame.locals import *

#main program begins
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("Circle Demo")
screen.fill((0,0,100))

pos_x = 300
pos_y = 250
radius = 200
```

```
angle = 360

#repeating loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]:
        sys.exit()

    #increment angle
    angle += 1
    if angle >= 360:
        angle = 0
        r = random.randint(0,255)
        g = random.randint(0,255)
        b = random.randint(0,255)
        color = r,g,b

    #calculate coordinates
    x = math.cos( math.radians(angle) ) * radius
    y = math.sin( math.radians(angle) ) * radius

    #draw one step around the circle
    pos = ( int(pos_x + x), int(pos_y + y) )
    pygame.draw.circle(screen, color, pos, 10, 0)

    pygame.display.update()
```

The Analog Clock Demo

It's really not much of a stretch to take the code from our Circle Demo and adapt it to make an analog clock. The difference will be that we need to draw lines from the center of the circle to the outer perimeter only where the hours, minutes, and seconds hands need to be positioned, based on the actual time of day. Let's learn how to do that.

Getting the Time

In Python, we use the `datetime` module (with an `import` statement) to gain access to the current time of day. Let's start by importing both the `date` and `time` modules from `datetime` to make our code a bit easier to write:

```
from datetime import datetime, date, time
```

Now, the key to getting the current date and time for our clock is a function called `datetime.today()`. Once we get a "snapshot" of the current date/time, then we can use the properties returned.

```
today = datetime.today()
```

The `today` variable will not contain the current date and time. Print it out from the Python prompt:

```
>>> today = datetime.today()
>>> today
datetime.datetime(2011, 6, 28, 16, 13, 29, 6000)
```

Each property is accessed by logical name: year, month, day, hour, minute, second, and microsecond. We can further segregate the date from the time like so:

```
>>> today.date()
datetime.date(2011, 6, 28)
>>> today.time()
datetime.time(16, 13, 29, 6000)
```

If we want *only* the time, we can just grab that by itself, although it doesn't hurt to have the date values as well:

```
>>> T = datetime.today().time()
>>> T
datetime.time(16, 20, 31, 295000)
```

At this point, the variable `T` contains the properties `T.hour`, `T.minute`, `T.second`, and `T.microsecond`. We can use these properties to make our clock program.

Drawing the Clock

First, we'll draw a large circle centered in the window, shown in Figure 5.7.

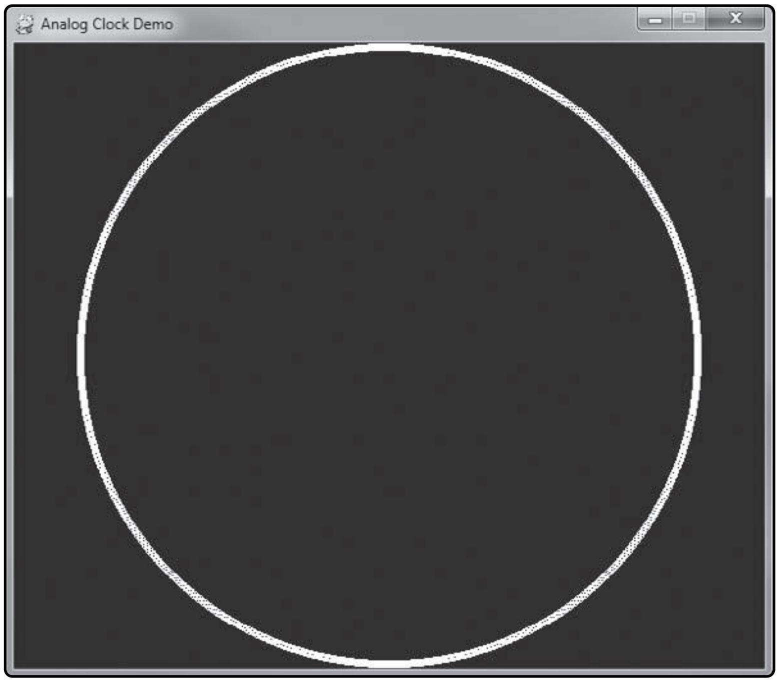


FIGURE 5.7

Getting started drawing the clock face.

```
pygame.draw.circle(screen, white, (pos_x, pos_y), radius, 6)
```

Numbers

Next, we'll draw the numbers around the clock, from 1 to 12. When drawing the number positions on the clock face, we can perform a simple calculation to find the position of each one. There are 360 degrees in a circle, and 12 numbers on a clock, so each number will be 360 degrees / 12 = 30 degrees apart. But, we have to account for the fact that angle 0 is pointing east, while the 12 o'clock position is north (from the center of the circle or clock face). So, we have to subtract 90 degrees when converting to radians. Figure 5.8 shows the clock at this stage, with the source code to follow.

TRAP

There's one obvious problem with our clock: the hours and minutes hands don't move partially in between the hour numbers as time goes on, they just jump from one number to the next (very much like a digital clock). It could be done by looking at the next-lower hand's position and adjusting based on the percentage that lower hand is around its cycle.

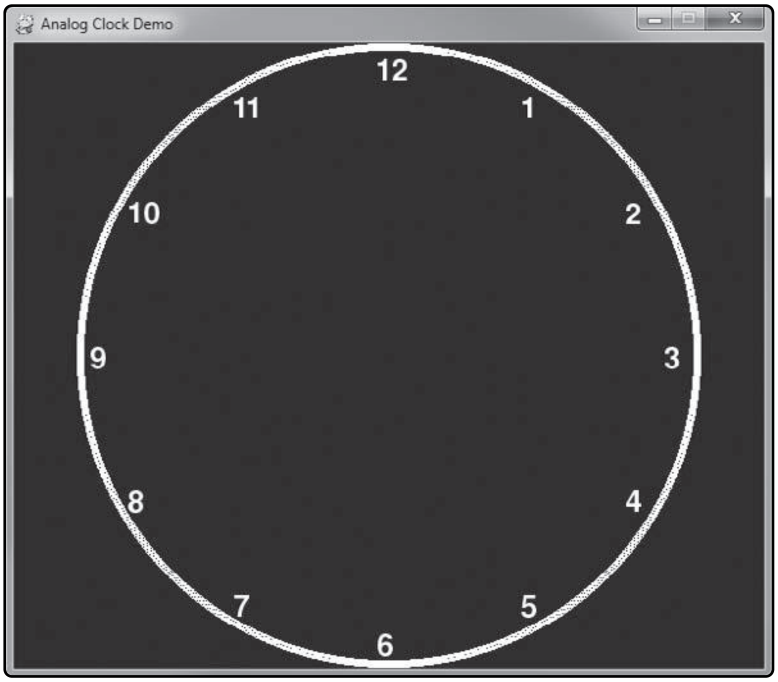


FIGURE 5.8

Drawing the numbered positions on the clock.

```
    for n in range(1,13):
        angle = math.radians( n * (360/12) - 90 )
        x = math.cos( angle ) * (radius-20) - 10
        y = math.sin( angle ) * (radius-20) - 10
        print_text(font, pos_x+x, pos_y+y, str(n))
```

Next, we'll draw the hour, minute, and second hands. The hours hand will be large, the minutes hand will be medium, and the seconds hand will be small. First, how do we rotate the hands so they point at the right number? That's easy! We have already learned the algorithm for it by drawing the numbered clock positions, which the hours hand will use. The minutes and seconds hands will be based on 60 rather than 12, so that will require a different calculation.

Hours

We get the current hour of the day with this line:

```
datetime.today().hour
```

The only problem is, the `hour` property is returned in 24-hour time format. Rather than dig into the `datetime` code for a conversion, let's just wrap it around 12-hour periods with the modulus character (%). Modulus keeps a value within a certain range. So, if you have a number, say, 15, but you want the limit to be 10, and have numbers wrap around, then

```
15 % 10 = 5
```

and that takes care of it without an `if` statement. Here's a solution that keeps the 24-hour values within a 12-hour time frame:

```
    today = datetime.today()
    hours = today.hour % 12
```

Drawing the hour hand in the right position requires a call to `pygame.draw.line()`. The first point of the line will be the center of the clock, and the second point will be near the correct number on the clock face corresponding to the current hour of the day. Figure 5.9 shows the hour hand pointing in the right direction. The conversion from hours to a degree angle is

```
hours * (360/12) - 90
```

(taking into account the correct adjustment for the starting point of the circle toward the right). The rest of the code is fine-tuning the position on the clock and proper wrapping of the angle.

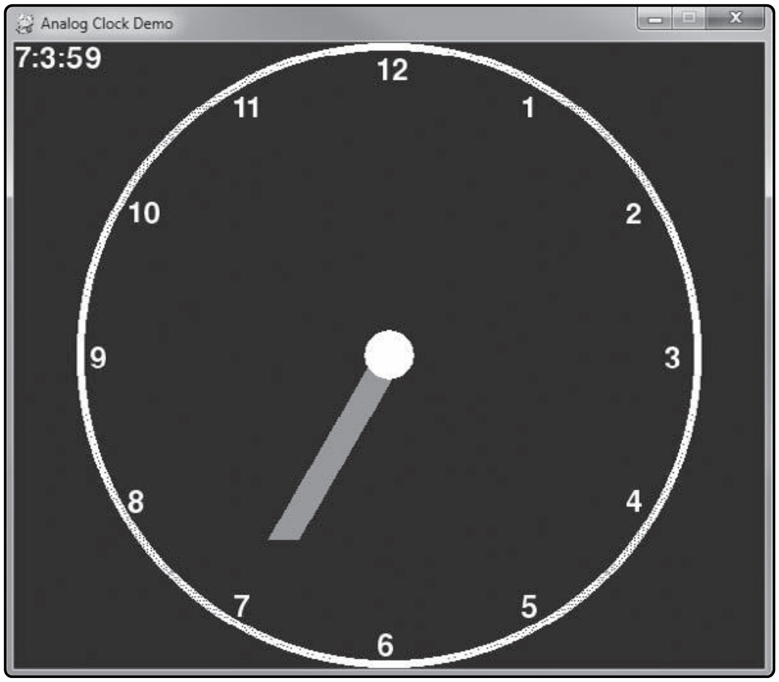


FIGURE 5.9
Drawing the HOUR hand on the clock.

```
    #draw the hours hand
    hour_angle = wrap_angle( hours * (360/12) - 90 )
    hour_angle = math.radians( hour_angle )
    hour_x = math.cos( hour_angle ) * (radius-80)
    hour_y = math.sin( hour_angle ) * (radius-80)
    target = (pos_x+hour_x,pos_y+hour_y)
    pygame.draw.line(screen, pink, (pos_x,pos_y), target, 25)
```

The helper function is called `wrap_angle()`. It accepts an angle in degrees and returns an angle (also in degrees) wrapped within a 360-degree circle. The function is beyond simple, but it helps a bit to clean up the code. If we have too many parentheses and inline conversions that makes the code hard to read.

```
def wrap_angle(angle):
    return abs(angle % 360)
```

Minutes

Calculating the position of the minute hand will be very similar to that of the hour hand, but we have to take into account 60 minutes in a complete hour, while the hours code was based on 12 segments. Figure 5.10 shows the result, with the code listing below.

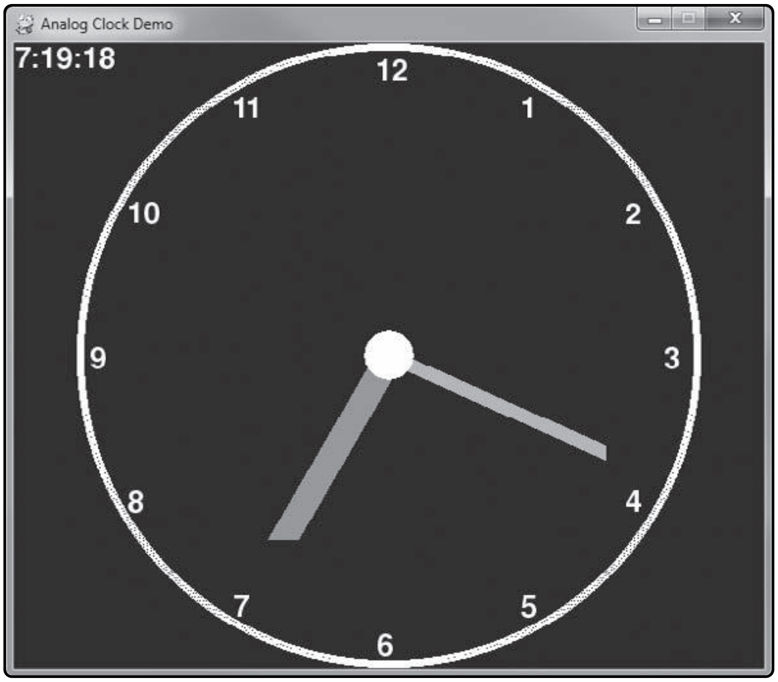


FIGURE 5.10

Drawing the MINUTE hand on the clock.

```
#draw the minutes hand
min_angle = wrap_angle( minutes * (360/60) - 90 )
min_angle = math.radians( min_angle )
min_x = math.cos( min_angle ) * (radius-60)
min_y = math.sin( min_angle ) * (radius-60)
target = (pos_x+min_x,pos_y+min_y)
pygame.draw.line(screen, orange, (pos_x,pos_y), target, 12)
```

Seconds

Seconds will be a duplication of the minute hand code, only taking the value from the seconds variable instead. The end result is shown in Figure 5.11, which is the finished Clock Demo!

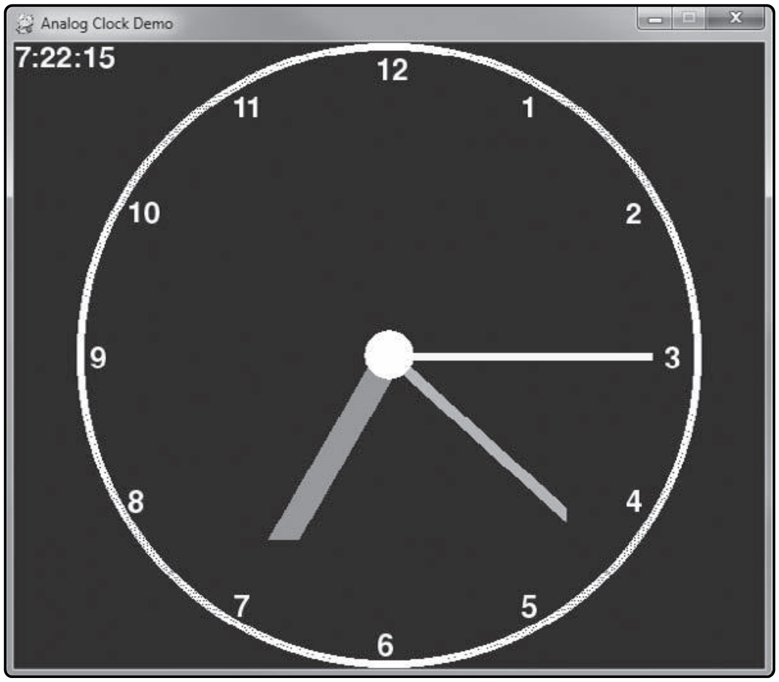


FIGURE 5.11
Drawing the SECOND hand on the clock.

```
#draw the seconds hand
sec_angle = wrap_angle( seconds * (360/60) - 90 )
sec_angle = math.radians( sec_angle )
sec_x = math.cos( sec_angle ) * (radius-40)
sec_y = math.sin( sec_angle ) * (radius-40)
target = (pos_x+sec_x,pos_y+sec_y)
```

```
    pygame.draw.line(screen, yellow, (pos_x,pos_y), target, 6)
```

Finished Code Listing

Just to be thorough, and due to the way the code has been tossed around in this chapter, following is the *complete* source code for the Clock Demo.

```
import sys, random, math, pygame
from pygame.locals import *
from datetime import datetime, date, time

def print_text(font, x, y, text, color=(255,255,255)):
    imgText = font.render(text, True, color)
    screen.blit(imgText, (x,y))

def wrap_angle(angle):
    return angle % 360

#main program begins
pygame.init()
screen = pygame.display.set_mode((600,500))
pygame.display.set_caption("Analog Clock Demo")
font = pygame.font.Font(None, 36)
orange = 220,180,0
white = 255,255,255
yellow = 255,255,0
pink = 255,100,100

pos_x = 300
pos_y = 250
radius = 250
angle = 360

#repeating loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]:
```

```
    sys.exit()

screen.fill((0,0,100))

#draw one step around the circle
pygame.draw.circle(screen, white, (pos_x, pos_y), radius, 6)

#draw the clock numbers 1-12
for n in range(1,13):
    angle = math.radians( n * (360/12) - 90 )
    x = math.cos( angle ) * (radius-20)-10
    y = math.sin( angle ) * (radius-20)-10
    print_text(font, pos_x+x, pos_y+y, str(n))

#get the time of day
today = datetime.today()
hours = today.hour % 12
minutes = today.minute
seconds = today.second

#draw the hours hand
hour_angle = wrap_angle( hours * (360/12) - 90 )
hour_angle = math.radians( hour_angle )
hour_x = math.cos( hour_angle ) * (radius-80)
hour_y = math.sin( hour_angle ) * (radius-80)
target = (pos_x+hour_x,pos_y+hour_y)
pygame.draw.line(screen, pink, (pos_x,pos_y), target, 25)

#draw the minutes hand
min_angle = wrap_angle( minutes * (360/60) - 90 )
min_angle = math.radians( min_angle )
min_x = math.cos( min_angle ) * (radius-60)
min_y = math.sin( min_angle ) * (radius-60)
target = (pos_x+min_x,pos_y+min_y)
pygame.draw.line(screen, orange, (pos_x,pos_y), target, 12)

#draw the seconds hand
sec_angle = wrap_angle( seconds * (360/60) - 90 )
```

```
    sec_angle = math.radians( sec_angle )
    sec_x = math.cos( sec_angle ) * (radius-40)
    sec_y = math.sin( sec_angle ) * (radius-40)
    target = (pos_x+sec_x,pos_y+sec_y)
    pygame.draw.line(screen, yellow, (pos_x,pos_y), target, 6)

    #cover the center
    pygame.draw.circle(screen, white, (pos_x,pos_y), 20)

    print_text(font, 0, 0, str(hours) + ":" + str(minutes) + ":" + str(seconds))

    pygame.display.update()
```

Summary

That concludes our chapter on math and graphics. We covered a lot of very important concepts in this chapter, of a variety that are found in nearly every video game, from a simple arcade-style game (such as *Peggle*) to a large, complex strategy game like *Command & Conquer 4*. In either case, we would find a lot of familiar code where movement, trajectories, and rotation of objects is concerned. In the next chapter, we'll be ramping it up another level by learning how to load and draw bitmaps, and then we'll use the code from *this* chapter to cause a spaceship to orbit around a planet.

Challenges

1. **The Circle Demo is the solution for numerous problems in a typical video game. To gain more experience with the relevant algorithm for moving around the perimeter of a circle, modify the program so that different shapes are drawn at each angle rather than a small filled circle.**
2. **The Analog Clock Demo is only just functional and barely passing on the cosmetic side of functionality. See if you can spruce it up with some better colors, perhaps a different background color, and different sizes for the numbers and clock hands.**

CHAPTER 6

Bitmap Graphics: The Orbiting Spaceship Demo

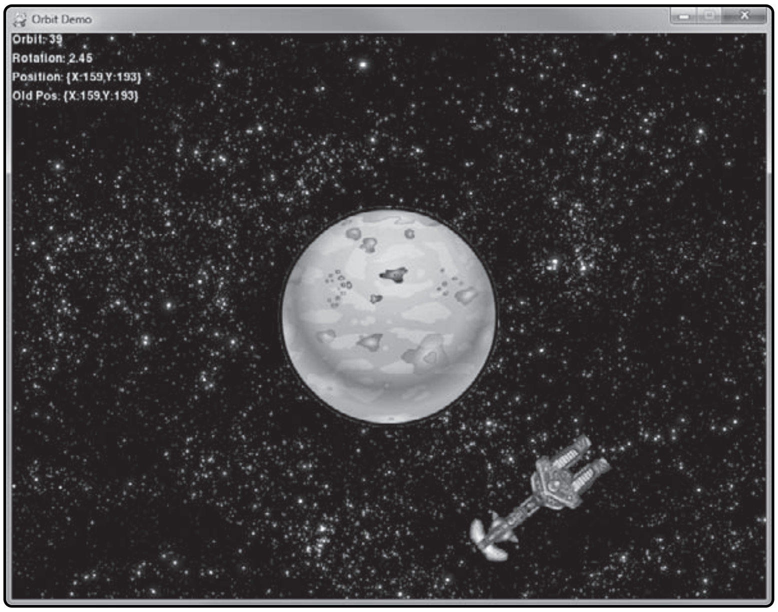
This chapter explains how to load and draw bitmaps using the `pygame.Surface` and `pygame.image` classes. We have already been using this class a bit and just taking it for granted up until now out of necessity. When the Pygame window is created with the call to `pygame.display.set_mode()`, a `Surface` object is returned, which we have called `screen` up to this point. Now we will learn more about this elusive `Surface` class and what its capabilities are between now and the following chapter, and really in every chapter from now on. Admittedly, we've done some interesting work with vector (line-based) graphics up to this point, but now it's time to study bitmaps, which is where you want to go for a good-looking game.

In this chapter we learn:

- How to load a bitmap
- How to draw a bitmap
- How to make a ship orbit a planet
- How to point an image in the right direction

Examining The Orbiting Spaceship Demo

The Orbiting Spaceship Demo shows how to use some of the math functions in Python to cause a spaceship to rotate around a planet, like the NASA Space Shuttle and ISS (International Space Station) orbits the Earth. The calculations are not actual acceleration-versus-gravity in nature, but just rotation of a point around a center point based on radius, but the end result looks the same and is good enough for a game. See Figure 6.1.



```
Orbit: 39
Rotation: 2.45
Position: {X:159,Y:193}
Old Pos: {X:159,Y:193}
```

FIGURE 6.1
The Orbiting Spaceship Demo.

USING BITMAPS

In Pygame, a bitmap is called a `Surface`. The "screen" object that we have been using until now with very little explanation is itself a `Surface` object (returned by the `pygame.display.set_mode()` function). Rather than demonstrate bitmap programming with several examples, we'll just get started on The Orbiting Spaceship Demo from the start and add to it as we go along.

Loading a Bitmap

First, let's learn how to load a bitmap, starting with the background image for the chapter demo. Pygame can handle quite a few bitmap file types via the `pygame.image.load()` function:

- JPG
- PNG
- GIF
- BMP

- PCX
- TGA
- TIF
- LBM, PBM, PGM, PPM, XPM

Our orbiting spaceship demo must have a background image of space, but I suppose just a black background would work too. Or, how about drawing random dots all over the background? You could do that with `pygame.gfxdraw.pixel()`! The `pygame.gfxdraw` module is based on the SDL drawing functions, which offer a few more shapes than `pygame.draw` has. Now let's just load a bitmap:

```
space = pygame.image.load("space.png").convert()
```

The trailing `convert()` function converts the image into the native color depth of the program window as an optimization. This really is required without exception. If you don't convert an image at load time, then it will be converted every time you draw it!

There's another variation of the function called `convert_alpha()` that you will want to use when loading foreground objects that have to be drawn with transparency. A TGA or PNG file can have alpha channel transparency in it, but some formats don't support it (like the older BMP format). If you just want to use `convert_alpha()` every time, even with images without transparency, there's no harm in doing that and it would be a bit more consistent.

TRICK

Be sure to use `Surface.convert_alpha()` when loading a bitmap with an alpha channel to tell Pygame you want to preserve the transparency in the image.

Drawing the Background

Drawing a bitmap is done with the `Surface` object, usually called `screen`, but it could be another `Surface` in memory, like a back buffer. We haven't covered double buffered drawing yet, but first things first, we're just now learning to draw a bitmap for the first time! To draw, use the `Surface` object. The `Surface` class has a function called `blit()` that draws a bitmap. The function name is short for "bit block transfer", a method of drawing by copying a chunk of memory from one location to another: from system memory to video memory, in this case. To draw the space bitmap starting at the upper-left corner:

```
    screen.blit(space, (0,0))
```

This is assuming the screen (i.e., window) has been initialized to a size that is large enough to hold the bitmap. I have used a size of 800,600. Here's our demo at this point (see Figure 6.2).

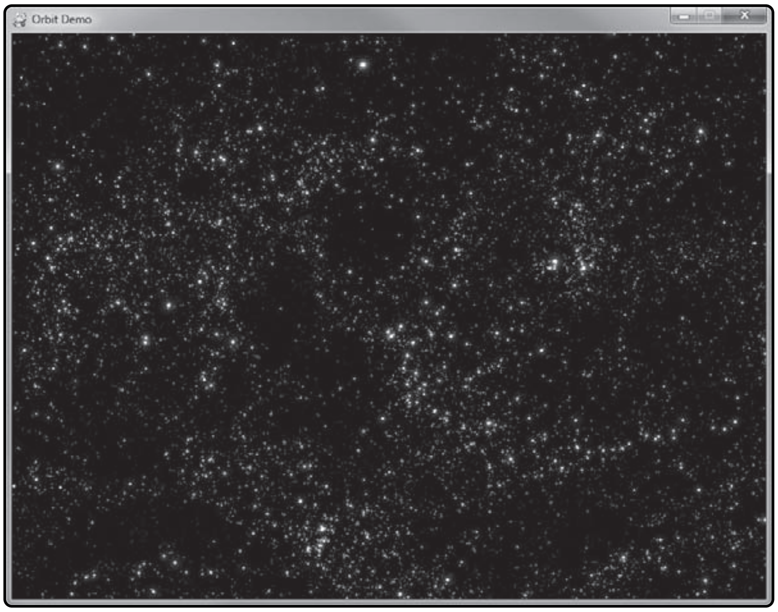


Figure 6.2

Drawing the background bitmap.

```
import random, math, pygame
from pygame.locals import *

#main program begins
pygame.init()
screen = pygame.display.set_mode((800,600))
pygame.display.set_caption("Orbit Demo")

#load bitmaps
space = pygame.image.load("space.png").convert()

#repeating loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]:
        sys.exit()
    #draw background
    screen.blit(space, (0,0))

    pygame.display.update()
```

HINT

The online reference manual for Pygame covers the `Surface` class in detail at this location: http://pygame.org/docs/ref/surface.html#pygame.Surface. I recommend keeping your web browser pointed here while learning bitmap programming as there will be some features here that you may find interesting but that are not covered in this chapter.

Drawing the Planet

Now we'll load and draw the planet image. Note that the artwork for these examples is found in the resource files for this and every chapter—a fact that is important now that we're relying on asset files that have to be loaded for our examples to work properly. First, let's load the planet before the `while` loop:

```
planet = pygame.image.load("planet2.png").convert_alpha()
```

Now, to draw the planet in this demo, we want to make it centered in the game window. Since the image dimensions could change (by someone editing the bitmap file), we would prefer to get the image dimensions in order to center it with code. This is better than "hard coding" the size of the bitmap. First, get the width and height of the bitmap using `Surface.get_size()`. Optionally, the width and height can be retrieved separately with `Surface.get_width()` and `Surface.get_height()`, respectively.

```
    width,height = planet.get_size()
    screen.blit(planet, (400-width/2,300-height/2))
```

In the code here, I have hard-coded the screen center but not the image. The screen's dimensions *could* change, but most likely this is something I've decided upon before working on the game. But, it is easy enough to get the center of the screen as well because it is also a `Surface` object. Figure 6.3 shows the planet.

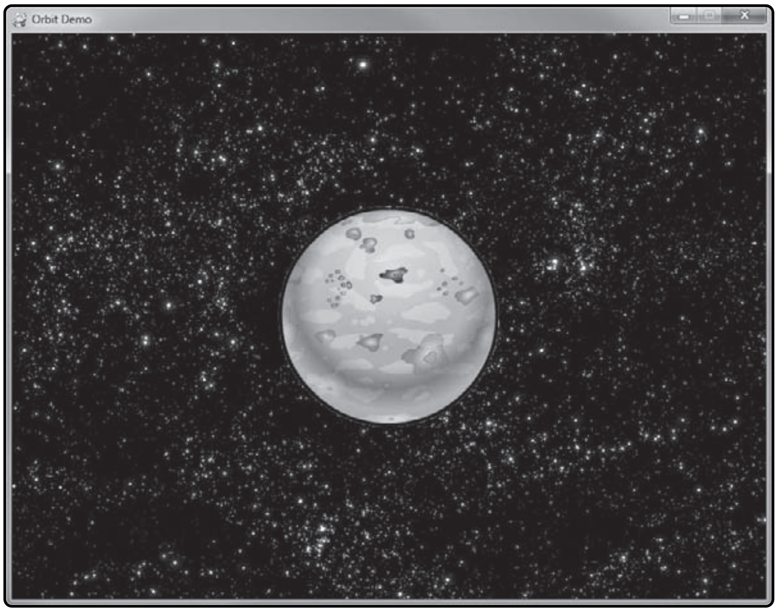


Figure 6.3

Drawing the planet bitmap transparently over the background.

Drawing the Spaceship

There are two spaceship bitmaps included with this chapter if you would like to use them for your own sci-fi themed games. The ships are quite nice looking, drawn by artist Ronald Conley for a game called *Starflight—The Lost Colony*. This game is free to download and play at www.starflightgame.com. The artwork is copyrighted but may be shared for non-commercial use. If you want to borrow any of the artwork from *Starflight* (or any other source!) for your own games, please give credit to the artist and source website to avoid legal problems. This is completely illegal with a commercial game, of course! Let's load the ship bitmap:

```
ship = pygame.image.load("freelance.png").convert_alpha()
```

The next line draws it, and the output is shown in Figure 6.4. Uh oh, the ship image is gigantic!

```
    screen.blit(ship, (50,50))
```

We could edit the bitmap with a graphic editor like Microsoft Paint, Paint.net, Gimp, or another similar tool. But, let's see if we can just shrink down the ship image with code instead. In order to do this, we have to sort of *cheat*. `Surface` has no means to change the scale of an image, so we have to shrink the spaceship by some other means. There is a class called `pygame.sprite.Sprite` that excels at drawing and manipulating images for use in a game, but that's a bit premature at this stage.

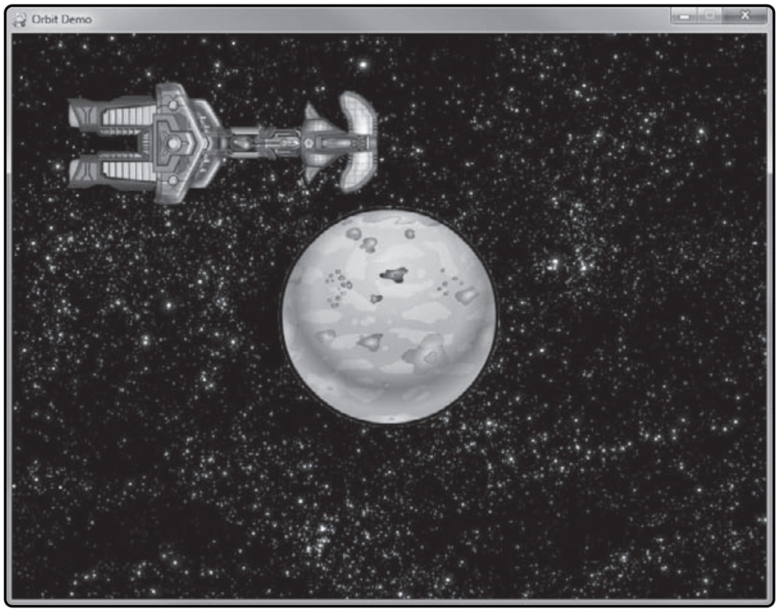


Figure 6.4

Drawing the spaceship bitmap.

Digging around in the Pygame docs (http://pygame.org/docs/ref/index.html), it turns out there is a module called `pygame.transform` that will meet our needs. This module has a bunch of helpful functions for working with images in creative ways, like scaling, flipping, and other things. First, let's look at `pygame.transform.scale()`—a *fast scaling* function that produces a quick scaled image but the pixels will look kind of *chunky*. Let's try it. This function is added right after the image is loaded. If you call this function inside the `while` loop it will just keep scaling the same image over and over until it's too tiny to see or too large to fit on the screen!

```
ship = pygame.image.load("freelance.png").convert_alpha()
width,height = ship.get_size()
ship = pygame.transform.scale(ship, (width//2, height//2))
```

Do you remember what the double division sign does in Python? It still does division, but it performs *integer division* rather than floating-point division. The result of this code is shown in Figure 6.5. It works! But, admittedly, the image is not very good.

So, let's try a better scaling function. There is a variation called `pygame.transform.smoothscale()`. This function takes more time to change the scale of the image because it over-samples the pixels and smoothes them out using one of two algorithms. For shrinking an image, like what we want to do, the pixels are averaged. For enlarging an image, a bilinear filter is used (a sort of blocky anti-aliasing technique). See Figure 6.6. The difference should be pretty clear even on the printed page, but if you want to really see clearly how the smooth version improves the appearance of the image, you'll need to open up the source code and run the program, changing the function call to see the difference.

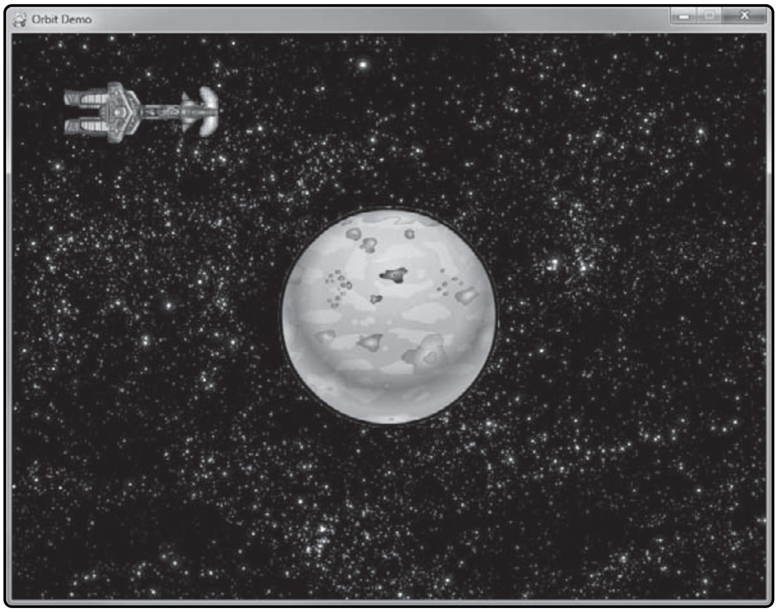


FIGURE 6.5

Drawing the spaceship bitmap scaled by 50%.

```
ship = pygame.transform.smoothscale(ship, (width//2,height//2))
```

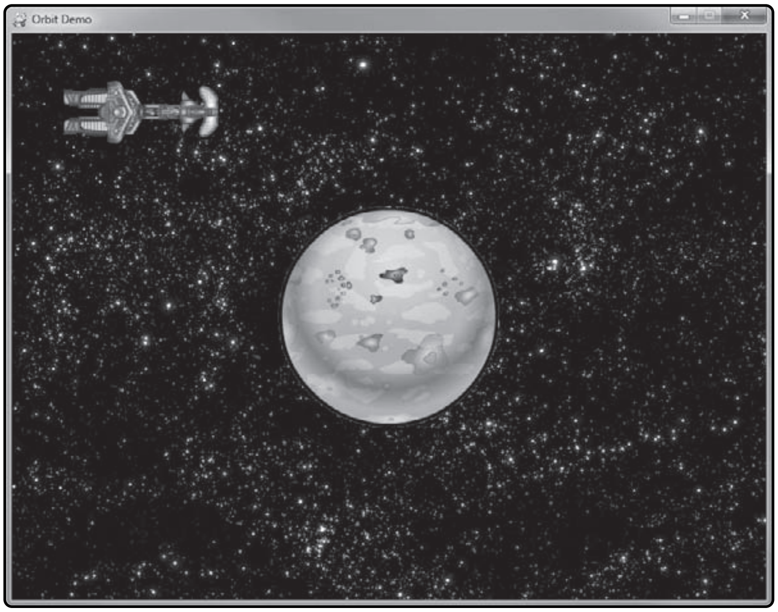


FIGURE 6.6

Scaling the spaceship bitmap with a better technique.

Orbiting the Planet

We have learned how to do basic bitmap drawing now, so we can use this new knowledge to make our demo. As you may recall from Chapter 5, “Math and Graphics: The Analog Clock Demo,” the trigonometry functions sine and cosine are used to draw circles and calculate trajectories. There’s a third function we haven’t used yet, tangent, that is useful in a similar but *tangential* way: pointing things in a certain direction. So, here’s what we want to do: make the spaceship orbit around the planet with sine and cosine, and then have it rotate so the front is always pointing in the direction it’s moving as it goes around the planet.

Orbiting

Let’s work on just getting the ship to orbit around the planet first. It will look kind of funny at first going around without changing orientation, although that is exactly how spaceships orbit planets. Making the ship’s nose always point in the direction it’s moving in orbit is totally not necessary, and not even realistic! But, for a video game, the usual player has certain expectations, and this is one of them—make the ship point in the direction it’s going. In some sci-fi movies, you might have noticed another thing they always do—keep the rocket engines firing constantly! That is also not done. Spaceships travel ballistically. This word is related to *shooting a gun* or *cannon*. Literally, a ship is *fired* and then it coasts along its path, just like a bullet or cannon ball. But, it just doesn’t look *cool*. If you want realistic, watch the movie *2001: A Space Odyssey*. Stanley Kubric got it right! Well, he *had* to, with the late great Sir Arthur C. Clark advising him on the making of the movie!

Based on the math code we learned in the previous chapter, we can cause the spaceship to move around any point on the screen at a certain radius. We’ll set that point at the center of the screen and rotate the ship around in a circle at a radius of 250 (based on a window size of 800,600). Now, there’s something we have to remember here, or else run into problems: the position is at the upper-left corner of the image, not at the center! So, when the ship is orbiting around the planet, we have to account for the ship size and adjust the position so that it is moving from the center of the ship image, not the upper-left corner.

Here’s a variation of the `Point` class introduced way back in Chapter 1 with some improvements: X and Y properties and an override of `__str__()` so the class data can be printed out with pre-coded formatting. Not familiar with Python properties? Well, this is a good time to learn how they work. Create a pair of “get” and “set” methods that return and set a private class variable. Then, using the desired name of the property (such as x or y), use the `property()` function to assign the “get” and “set” methods associated with that class variable. The benefit of a property over just using a global variable is the ability to control its bounds while keeping the code clean.

```
class Point(object):
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    #X property
    def getx(self):
        return self.__x
    def setx(self, x):
        self.__x = x
    x = property(getx, setx)

    #Y property
    def gety(self):
        return self.__y
    def sety(self, y):
        self.__y = y
    y = property(gety, sety)

    def __str__(self):
        return "{X:" + "{:.0f}".format(self.__x) + \
            ",Y:" + "{:.0f}".format(self.__y) + "}"
```

Putting the `Point` class to work, we need two instances in our program:

```
pos = Point(0,0)
old_pos = Point(0,0)
```

Next, here's how we'll move the ship in its "orbit":

```
    angle = wrap_angle(angle - 0.1)
    pos.x = math.sin( math.radians(angle) ) * radius
    pos.y = math.cos( math.radians(angle) ) * radius
```

Here's the code to draw the ship, taking into account the image size:

```
    width,height = ship.get_size()
    screen.blit(ship, (400+pos.x-width//2,300+pos.y-height//2))
```

The current version of the demo now with the ship orbiting is shown in Figure 6.7.

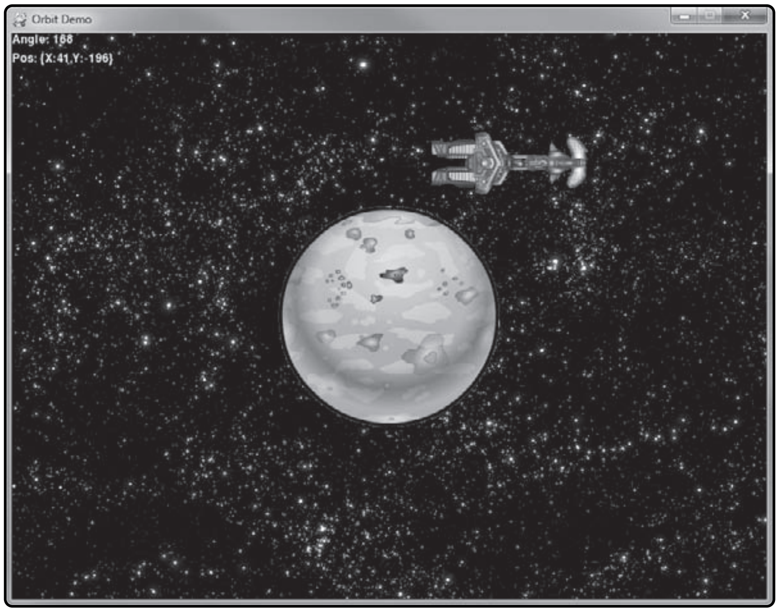


FIGURE 6.7
The spaceship is now rotating around the planet.

Rotating

So far, so good! The ship now will need to be rotated so that it's pointing in the direction it's moving around the planet. This is going to be a little tricky. There's a little-known math function that's pure magic! It's called `math.atan2()`, and is a function that calculates arc-tangent with two parameters. We pass to this function two parameters: `delta_y` and `delta_x`. These delta values represent the *difference* between the X and Y properties of two coordinates on the screen. Almost as if by magic, the resulting value returned by `math.atan2()` is the *angle* to the target! All we do after that is rotate the image to that target angle and it will appear to point in the direction it's moving.

Now for the tricky part. How do we know where the spaceship image is going to be in the *next* frame while the demo is running? By making a prediction! We can write code to *predict* where the ship will be in the *future*! Here's the magic algorithm: Keep track of the *last* position of the ship; then use `math.atan2()` using the *current* and *last* position; then add 180 degrees to the resulting angle returned by `math.atan2()`. Do you see how that works? We get the angle to the *previous* position of the ship moments ago, and rotate the ship to that angle, but flip it around 180 degrees, completely backward from that angle, and presto, that is where the ship is *heading*! This is another one of those phenomenally awesome functions that is used all the time in game development for all sorts of things!

Let's put `math.atan2()` to work. We'll need a

```
delta_x = ( pos.x - old_pos.x )
delta_y = ( pos.y - old_pos.y )
rangle = math.atan2(delta_y, delta_x)
rangled = wrap_angle( -math.degrees(rangle) )
```

I've used the *rangle* variable to represent the radian angle calculated by `math.atan2()`, and the *rangled* variable is the angle converted to degrees and wrapped. Once the angle is available, then we can rotate the ship image to the desired angle. This requires the `pygame.transform` module again. It's a pretty useful module, as you can see! The function we need is `pygame.transform.rotate()`, with the source image and desired rotation angle as parameters, and a new image returned. A scratch variable is used for the new image.

```
scratch_ship = pygame.transform.rotate(ship, rangled)
```

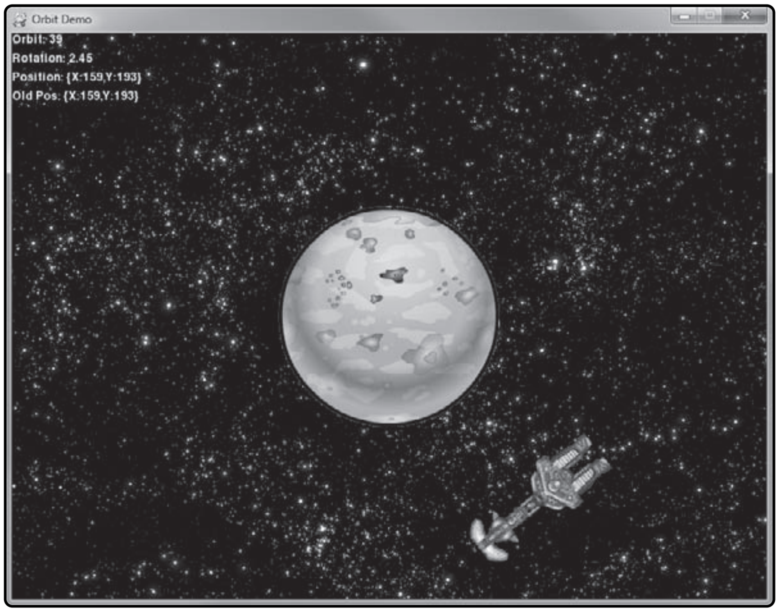
Now we can draw the ship. But we can't use the *original* ship image that hasn't changed; we have to use the new image called `scratch_ship`, for both the position calculation *and* drawing. Note in the code that follows that the `scratch_ship` image is used to get the width and height. `Surface.get_size()` calculates the width and height of the rotated image in this case.

```
width,height = scratch_ship.get_size()
x = 400+pos.x-width//2
y = 300+pos.y-height//2
screen.blit(scratch_ship, (x,y))
```

After everything else, all we have to do is "remember" the position of the ship for use next time through the `while` loop (also called "the next frame" in game parlance).

```
old_pos.x = pos.x
old_pos.y = pos.y
```

Figure 6.8 shows the finished program. The complete code listing follows for reference (less the `Point` class, which was already shown in its entirety).



```
Orbit: 39
Rotation: 2.45
Position: {X:159,Y:193}
Old Pos: {X:159,Y:193}
```

Figure 6.8

The spaceship rotates as it orbits the planet.

```
import sys, random, math, pygame
from pygame.locals import *

#Point class definition goes here . . .

#print_text function
def print_text(font, x, y, text, color=(255,255,255)):
    imgText = font.render(text, True, color)
    screen.blit(imgText, (x,y))

#wrap_angle function
def wrap_angle(angle):
    return angle % 360

#main program begins
pygame.init()
screen = pygame.display.set_mode((800,600))
pygame.display.set_caption("Orbit Demo")
font = pygame.font.Font(None, 18)
```

```
#load bitmaps
space = pygame.image.load("space.png").convert_alpha()
planet = pygame.image.load("planet2.png").convert_alpha()
ship = pygame.image.load("freelance.png").convert_alpha()
width,height = ship.get_size()
ship = pygame.transform.smoothscale(ship, (width//2, height//2))

radius = 250
angle = 0.0
pos = Point(0,0)
old_pos = Point(0,0)

#repeating loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]:
        sys.exit()

    #draw background
    screen.blit(space, (0,0))

    #draw planet
    width,height = planet.get_size()
    screen.blit(planet, (400-width/2,300-height/2))

    #move the ship
    angle = wrap_angle(angle - 0.1)
    pos.x = math.sin( math.radians(angle) ) * radius
    pos.y = math.cos( math.radians(angle) ) * radius

    #rotate the ship
    delta_x = ( pos.x - old_pos.x )
    delta_y = ( pos.y - old_pos.y )
    rangle = math.atan2(delta_y, delta_x)
    rangled = wrap_angle( -math.degrees(rangle) )
    scratch_ship = pygame.transform.rotate(ship, rangled)

    #draw the ship
    width,height = scratch_ship.get_size()
    x = 400+pos.x-width//2
    y = 300+pos.y-height//2
    screen.blit(scratch_ship, (x,y))

    print_text(font, 0, 0, "Orbit: " + "{:.0f}".format(angle))
    print_text(font, 0, 20, "Rotation: " + "{:.2f}".format(rangle))
    print_text(font, 0, 40, "Position: " + str(pos))
    print_text(font, 0, 60, "Old Pos: " + str(old_pos))

    pygame.display.update()

    #remember position
    old_pos.x = pos.x
    old_pos.y = pos.y
```

Summary

This chapter was a fun romp through more rocket science and the addition of the bitmap features of Pygame! There is just no comparison between vector shapes and bitmap graphics. As The Orbiting Spaceship Demo in this chapter demonstrated, we can do a lot with bitmaps and some interesting math functions, and we haven't even touched upon sprite programming yet! That's coming up in the very next chapter.

Challenges

1. **Replace the current spaceship image in the orbiting demo with the additional spaceship bitmap provided with this chapter.**
2. **Modify the program so that pressing the + and - keys will cause the spaceship to orbit faster or slower around the planet, respectively.**
3. **Using the formula for the circumference of a circle, calculate the distance travelled by the ship in one complete orbit based on its radius and display the answer on the screen.**

CHAPTER 7

Animation with Sprites: The Escape the Dragon Game

The previous chapter was a pretty good introduction to bitmap graphics programming. We learned that Pygame has a lot of good features for working with bitmaps. But, aside from the capabilities of the `pygame.transform` module, which includes scaling and rotation of bitmaps, there is no practical way to do animation with it. This is where the `pygame.sprite` module takes over, and that is the subject we're learning about in this chapter.

In this chapter, you will learn to:

- Manually animate a sprite with a special calculation
- Use features in the `pygame.sprite` module
- Make a game called Escape the Dragon!

Examining The Escape The Dragon Game

The sample game in this chapter will help you to understand sprite programming with Python and Pygame. The premise of the game is very simple: a dragon is chasing your character, so you must jump over flaming arrows coming toward you so that they hit the dragon and stop it from chasing you. The concept comes from one of the mini-games in the Facebook game *Ninja Wars*.

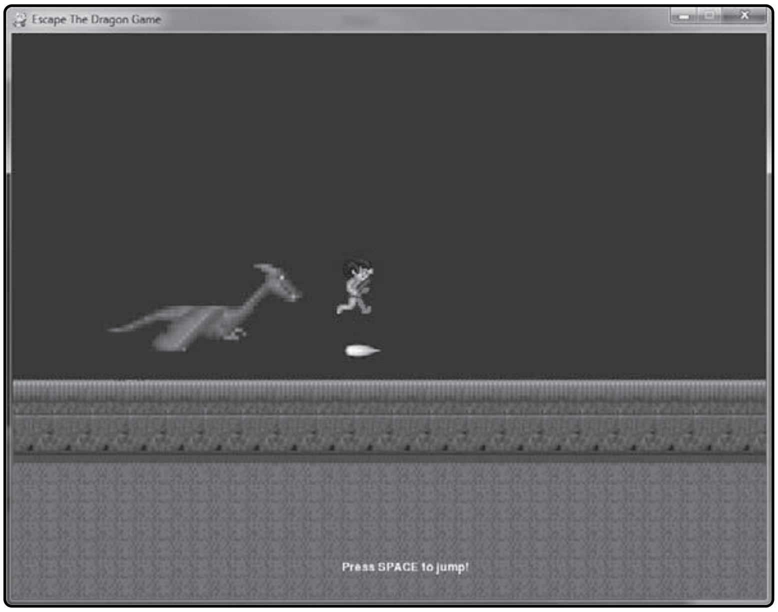


Figure 7.1

The Escape the Dragon Game features animated sprites.

Using Pygame Sprites

The `pygame.sprite` module contains a class called `Sprite` that we can use as a *starting point* for our game sprites. I say *starting point* because `pygame.sprite.Sprite` is not a complete solution, it's just a limited class that knows how to work with groups to update and draw itself. Even that is a bit of a stretch, given that we have to write the code to do these things. From the most objective point of view, a Pygame sprite contains an image (`image`) and a position (`rect`). We have to extend this with our own class to provide the features we want in a fully functional game sprite class.

Custom Animation

Animation with Pygame is a bit tricky, only because we have to know how `pygame.sprite.Sprite` works in order to write our own animation code. A Pygame sprite will be based around its `image` and `rect` properties, as already mentioned, so the trick is to wrap animation code around these two properties. When that is done, then the sprite group will automatically update the animation frame image and draw the specific frame (rather than the whole sprite sheet image). Let's look at a sprite sheet image first to get an idea how this is going to work. Figure 7.2 shows just such an image.

HINT

The animated dragon sprite was drawn by Ari Feldman (www.flyingyogi.com). You can download his collection of free game sprites, called SpriteLib, from his website.

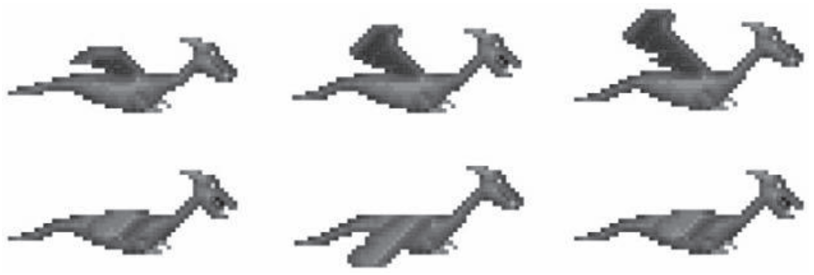


FIGURE 7.2

The dragon sprite image has six frames of animation. Courtesy of Ari This was cutoff on my printout.

A sprite sheet image contains rows and columns of “tiles” or “frames,” each of which is one frame of the animation sequence. Figure 7.3 shows one frame highlighted in a sprite sheet with the rows and columns labeled for easier reference. Note that they are 0-based! This is important, as the calculations rely on counting the frame number starting at zero, not one.

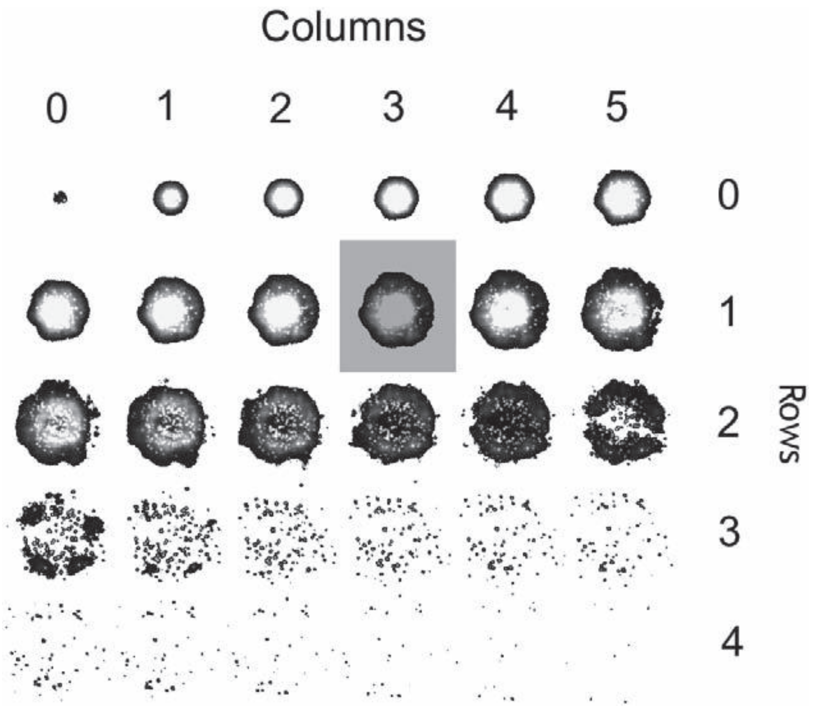


FIGURE 7.3

Illustration of the rows and columns in a sprite sheet image.

The sprite sheet images shown here will be loaded and retained as the master image used for animation. While a sprite is being moved and drawn in a game, a call to the `update()` method will be made by the sprite group automatically, as is the call to `draw()`. We can write our own `update()` method, but `draw()` is not replaced, it's passed on to the parent `pygame.sprite.Sprite.draw()` method. What we must do is make sure the image property of `pygame.sprite.Sprite` contains the image of the current *frame* of the animation, not the whole sprite sheet. Because of the way this works, the sprite sheet (master image) will be loaded as an independent class variable, not loaded directly into `Sprite.image`.

Loading a Sprite Sheet

When we load the master image, we must tell our sprite class how large one frame is—that is, the width and height of a single frame are passed as parameters when a new sprite is created. Usually the most sensible name for the method is `load()`, and typically it will have a filename parameter. In addition to frame width and height, we must also tell our sprite class how many *columns* there are in the sprite sheet. Take a look at the illustration in Figure 7.3 again for reference. Note that the highlighted frame is under column 3. This is really all we need to know, the number of columns, because the number of rows does not matter in the calculation to draw a single frame.

Let's try writing a function that will get the job done of loading an image and setting a sprite's properties. The function definition below requires a filename, width, height, and columns as parameters. These are the bare essentials for doing sprite animation. We'll peruse a complete class listing after going over the theory behind these concepts, so don't worry about typing in any of this code just yet.

```
def load(self, filename, width, height, columns):
    self.master_image = pygame.image.load(filename).convert_alpha()
    self.frame_width = width
    self.frame_height = height
    self.rect = 0,0,width,height
    self.columns = columns
```

Changing the Frame

Normally, animation proceeds one frame at a time from first to last. A more advanced animation system will allow a sprite to animate forward, backward, and within any specified range of the animation set. We'll keep it simple by just animating from first to last frame, then wrapping around to the first frame again. This is pretty easy to write in code:

```
    self.frame += 1
    if self.frame > self.last_frame:
        self.frame = self.first_frame
```

The trick is not so much changing the frame number, but making that happen at a certain *time interval*. Yes, we have to use timing code! It's a bit of a challenge to wrap your mind around at first. At least, it was for me! But once you learn the basic Python code for getting the current time value in ticks, then the rest of the process is pretty easy to handle.

First, we need to create an object variable from `pygame.time.Clock()`. I have called my variable `framerate`:

```
framerate = pygame.time.Clock()
```

When this `Clock()` method is called, it starts an internal timer running from that point forward that we can use to get incremental time update values, with even the option to set the game running at a fixed framerate. Inside the main `while` loop in a game, then, call:

```
    framerate.tick(30)
```

The parameter, `30`, can be set to any desired framerate. It does a pretty good job of keeping the game running at this speed, but 30 might be too slow for some games that would run better at 40 or 60 (the most common framerates used).

That's the first step, just to get the game loop running at a consistent framerate. Next, we need a timing variable that works not at the speed of framerates, but at the millisecond level. The `pygame.time` module has a method called `get_ticks()` that will meet our needs for the purpose of timing sprite animation.

```
    ticks = pygame.time.get_ticks()
```

This `ticks` variable can be passed to our own sprite class' `update()` method to give our sprites independent animation timing at any desired framerate. In the code below, note that unless the timing is correct, the animation frame does not change.

```
    def update(self, current_time, rate=0):
        if current_time > self.last_time + rate:
            self.frame += 1
            if self.frame > self.last_frame:
                self.frame = self.first_frame
            self.last_time = current_time
```

In addition to the animation frame update code shown here, we also will be copying the current frame *image* into `Sprite.image` (`self.image`, in this case), which is the subject of the next paragraph.

Drawing One Frame

Knowing that `Sprite.draw()` is called automatically by the sprite group, we will not be writing our own *drawing* code, only setting up the properties to make the draw happen the way we want it to. That is done in the `update()` method of our custom sprite class. `Sprite.draw()` expects that `Sprite.image` and `Sprite.rect` are set to valid values or else an error will occur (a common error is an invalid position when the `rect` is undefined).

To draw a single frame from a sprite sheet, we must calculate the X,Y position of the frame's top-left corner, and then copy the frame image based on the frame's width and height. The X position represents the column number. The Y position represents the row number. We calculate Y, or row, by dividing the frame number by the number of columns, and then multiplying that value by the frame height:

```
Y = (frame / columns) * height
```

To calculate the X, or column value, we divide frame by columns again, but this time we only care about the *remainder*, not the quotient (in mathematical terms, that is the answer to a division problem). We can get the remainder by using modulus rather than division, and then multiply the value by the frame width:

```
X = (frame % columns) * width
```

These formulas can be written using Python code to update the `Sprite.image` used to draw a single frame. To get the frame image out of the sprite sheet, `Surface.blit()` could be used, but there's a far easier way. Using the X and Y position values along with the frame width and height, we can just create a `Rect` and pass it to a different, and rather interesting method called `subsurface()`. This does not actually *copy* or *blit* the image at all, it just sets up a pointer to the existing master image! So, in effect, we're going to be doing lightning-fast updates of the frame image because no pixels have to be copied at all!

```
    frame_x = (self.frame % self.columns) * self.frame_width
    frame_y = (self.frame // self.columns) * self.frame_height
    rect = ( frame_x, frame_y, self.frame_width, self.frame_height )
    self.image = self.master_image.subsurface(rect)
```

TRICK

Always be on the lookout for awesome coding tricks like using `Surface.subsurface()` rather than drawing copies of every frame into an array or collection! By doing it this way, the code is greatly simplified, *and* there's no performance hit!

Sprite Groups

Pygame uses sprite *groups* to manage updating and drawing sprites, as a means to handle a large number of entities usually found in a typical game. This is a good idea, as it saves us the trouble of doing it manually. It's odd, though, that the Pygame creators thought to include an iterated sprite entity manager but did not include even rudimentary animation support. No matter, we'll use what is provided and add our own code as needed!

A sprite group is a simple entity container that will call a sprite class' `update()` method with whatever parameters it supports, and then draw all sprites contained in the container. A sprite group is created with `pygame.sprite.Group()` like so:

```
group = pygame.sprite.Group()
group.add(sprite)
```

where the *`sprite`* parameter is a sprite object that has already been created. After creating a group, any number of sprites can be added to the group container so that they can be managed more easily, and this also cuts down on global variable use. When we're ready to update and draw the sprites in our game, we do this entirely with the group rather than the individual sprites:

```
    group.update(ticks)
    group.draw(screen)
```

The real power here is not containing all game sprites inside one group and using it to manage them, but creating *several* groups for each *type* of game sprite! This allows custom behaviors to be applied to specific types of sprites managed by their own group container objects. Another great advantage to using groups is that the updating and drawing code need not change when game objects are added or removed—the same `update()` and `draw()` methods are called and the group updates all of its attached sprite objects.

TRAP

Be careful not to accidentally overwrite the base `Sprite.rect` property with a basic tuple. That's an easy mistake to make! Always set `Sprite.rect` to a new `Rect()`, like `Rect(0,0,100,100)`, and not just an undefined tuple like `(0,0,100,100)`. Python allows you to do that, and it can create the most bizarre error messages when things are expecting `Sprite.rect` to be a `Rect`, but it's been replaced with a tuple! This is confusing because some rectangle code still works with the tuple-ized version!

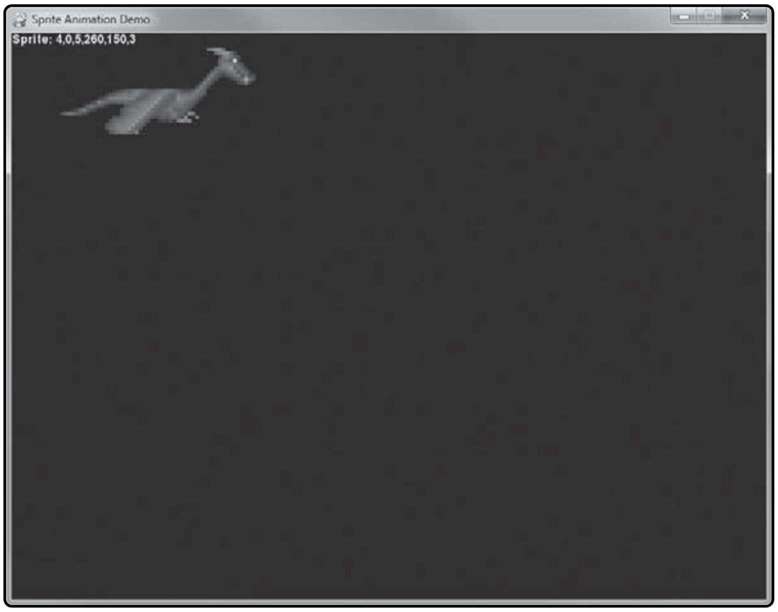
MySprite Class

We can put all of this code into a reusable class which I'll just call `MySprite` for lack of a better name. This class directly inherits (that is, extends) `pygame.sprite.Sprite`, and works directly with `pygame.sprite.Group`, for automated update and drawing. There are quite a few properties in this enhanced sprite class called `MySprite`, properties dealing with the animation, the master image, and so forth, that are not already in the base sprite class. But also, our new `MySprite` class is not overly complex, not filled with complex methods or properties, so think of this class as just a starting point for your own future sprite programming work. This is intentionally simple, a skeleton class for working with animated sprites. There are also three properties in the `MySprite` class: X, Y, and position. These are meant to help with setting the position of the sprite. Without these properties, we have to modify the `rect` which is kind of a pain when you just want to change the X or Y value.

```
class MySprite(pygame.sprite.Sprite):
    def __init__(self, target):
        pygame.sprite.Sprite.__init__(self) #extend the base Sprite class
        self.master_image = None
        self.frame = 0
        self.old_frame = -1
        self.frame_width = 1
        self.frame_height = 1
        self.first_frame = 0
        self.last_frame = 0
        self.columns = 1
        self.last_time = 0

    #X property
    def _getx(self): return self.rect.x
    def _setx(self,value): self.rect.x = value
    X = property(_getx,_setx)

    #Y property
    def _gety(self): return self.rect.y
    def _sety(self,value): self.rect.y = value
    Y = property(_gety,_sety)

    #position property
    def _getpos(self): return self.rect.topleft
    def _setpos(self,pos): self.rect.topleft = pos
    position = property(_getpos,_setpos)

    def load(self, filename, width, height, columns):
        self.master_image = pygame.image.load(filename).convert_alpha()
        self.frame_width = width
        self.frame_height = height
        self.rect = Rect(0,0,width,height)
        self.columns = columns
        #try to auto-calculate total frames
        rect = self.master_image.get_rect()
        self.last_frame = (rect.width // width) * (rect.height // height) - 1

    def update(self, current_time, rate=30):
        #update animation frame number
        if current_time > self.last_time + rate:
            self.frame += 1
            if self.frame > self.last_frame:
                self.frame = self.first_frame
            self.last_time = current_time

        #build current frame only if it changed
        if self.frame != self.old_frame:
            frame_x = (self.frame % self.columns) * self.frame_width
            frame_y = (self.frame // self.columns) * self.frame_height
            rect = Rect(frame_x, frame_y, self.frame_width, self.frame_height)
            self.image = self.master_image.subsurface(rect)
            self.old_frame = self.frame

    def __str__(self):
        return str(self.frame) + "," + str(self.first_frame) + \
               "," + str(self.last_frame) + "," + str(self.frame_width) + \
               "," + str(self.frame_height) + "," + str(self.columns) + \
               "," + str(self.rect)
```

Sprite Animation to the Test

Figure 7.4 shows the output of the Sprite Animation Demo program, and the code listing follows. The `MySprite` class was just listed above, so it's not repeated here; just be sure to include it in the program's source code before trying to run the program.



```
Sprite: 4,0,5,260,150,3
```

Figure 7.4

The Sprite Animation Demo program.

```
import pygame
from pygame.locals import *

# remember to include MySprite here

#print_text function
def print_text(font, x, y, text, color=(255,255,255)):
    imgText = font.render(text, True, color)
    screen.blit(imgText, (x,y))

#initialize pygame
pygame.init()
screen = pygame.display.set_mode((800,600),0,32)
pygame.display.set_caption("Sprite Animation Demo")
font = pygame.font.Font(None, 18)
```

```
framerate = pygame.time.Clock()

#create the dragon sprite
dragon = MySprite(screen)
dragon.load("dragon.png", 260, 150, 3)
group = pygame.sprite.Group()
group.add(dragon)

#main loop
while True:
    framerate.tick(30)
    ticks = pygame.time.get_ticks()

    for event in pygame.event.get():
        if event.type == pygame.QUIT: sys.exit()
    key = pygame.key.get_pressed()
    if key[pygame.K_ESCAPE]: sys.exit()

    screen.fill((0,0,100))
    group.update(ticks)
    group.draw(screen)
    print_text(font, 0, 0, "Sprite: " + str(dragon))
    pygame.display.update()
```

The Escape the Dragon Game

Now we'll use the `MySprite` class and the new sprite animation code just studied to create a simple game to demonstrate how to use the new class. Figure 7.5 shows one of the two ways to finish the game, and this does not bode well for the caveman character!

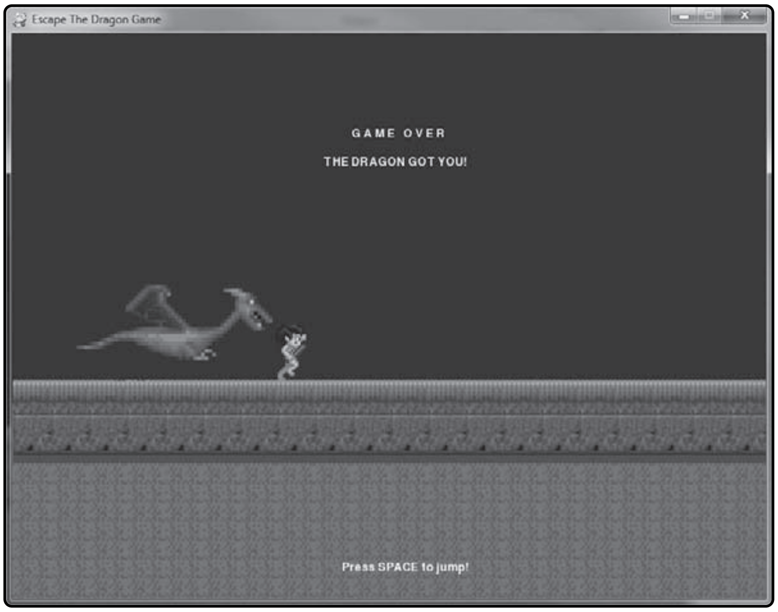


FIGURE 7.5

This caveman did not escape the dragon!

Jumping

The gameplay is simple—jump over the flaming arrows so they will hit the dragon, and escape! The Space key is used to jump over the arrows. The way this works is very simple in concept, but a bit challenging to understand at first. A Y velocity value is set to a negative number like -8.0. While the player is in "jumping mode," so to speak, the velocity is *increased* by a small amount every frame. In this game, we want the player's caveman sprite to jump up *quickly* but also fall back down quickly to make the game challenging. So, the modifier is a value of 0.5 (added to the velocity every frame). You can see the player jumping over a flaming arrow by flipping back to Figure 7.1 again. The end result is as follows:

-8.0 + 0.5 = -7.5

-7.5 + 0.5 = -7.0

-7.0 + 0.5 = -6.5

-6.5 + 0.5 = -6.0

-6.0 + 0.5 = -5.5

and so on, until we reach:

-0.5 + 0.5 = 0.0

At this point, the sprite will have peaked at the top of the jump and will begin moving *back down* again toward the ground:

0.0 + 0.5 = 0.5

0.5 + 0.5 + 1.0

1.0 + 0.5 = 1.5

and so on until the sprite reaches the starting Y position, at which point the jump cycle ends and the velocity is no longer used. When the player presses Space to jump again, that velocity value is restarted at -8.0 again. You can experiment with different heights by adjusting this and the incremental value to tweak the gameplay. Figure 7.6 shows what happens when the dragon is hit by enough arrows that it is pushed off the screen. That's how to win, by jumping over the arrows so they hit the dragon!

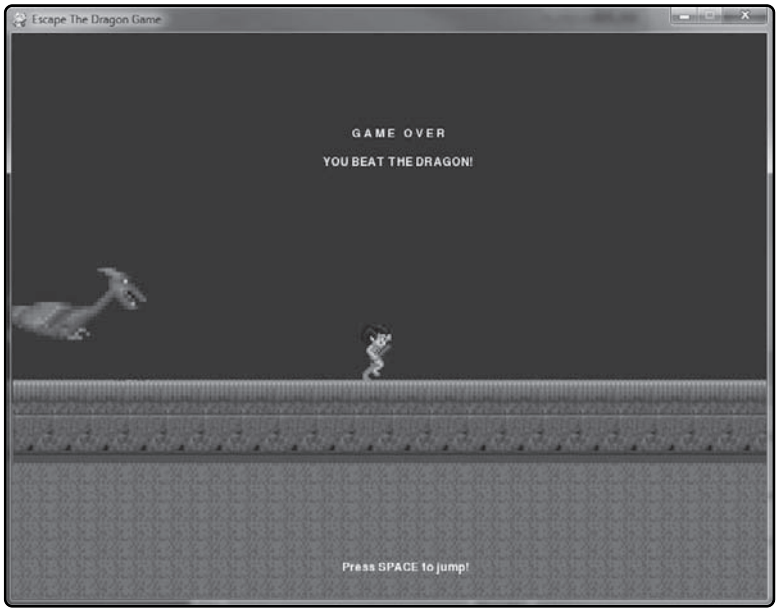


Figure 7.6

The dragon was pushed off the screen by the flaming arrows.

Colliding

We haven't covered sprite collisions yet, and won't get into it in detail until the next chapter, so a quick perusal is in order now. There are several functions we can use to detect when two sprites collide with each other, using either the so-called "bounding rectangle" technique or "bounding circle." Bounding rectangle collision detection works by comparing the rectangles of two sprites to see if they overlap. That is the technique used in this game to determine when the arrow has hit the player or the dragon, or when the dragon has "eaten" the player. Here is one example, comparing the arrow with the dragon sprite to see if there's a collision:

```
pygame.sprite.collide_rect(arrow, dragon)
```

As long as we have inherited from `pygame.sprite.Sprite` for our own sprite class, then the `rect` property will be available, as that is what `pygame.sprite.collide_rest()` uses to see if the two sprites have hit each other.

Source Code

Here is the source code for The Escape the Dragon Game. I hope you enjoy it.

```
import sys, time, random, math, pygame
from pygame.locals import *

# insert MySprite class definition here

def print_text(font, x, y, text, color=(255,255,255)):
    imgText = font.render(text, True, color)
    screen.blit(imgText, (x,y))

def reset_arrow():
    y = random.randint(250,350)
    arrow.position = 800,y

#main program begins
pygame.init()
screen = pygame.display.set_mode((800,600))
pygame.display.set_caption("Escape The Dragon Game")
font = pygame.font.Font(None, 18)
framerate = pygame.time.Clock()

#load bitmaps
bg = pygame.image.load("background.png").convert_alpha()

#create a sprite group
group = pygame.sprite.Group()

#create the dragon sprite
dragon = MySprite(screen)
```

```
dragon.load("dragon.png", 260, 150, 3)
dragon.position = 100, 230
group.add(dragon)

#create the player sprite
player = MySprite(screen)
player.load("caveman.png", 50, 64, 8)
player.first_frame = 1
player.last_frame = 7
player.position = 400, 303
group.add(player)

#create the arrow sprite
arrow = MySprite(screen)
arrow.load("flame.png", 40, 16, 1)
arrow.position = 800,320
group.add(arrow)

arrow_vel = 8.0
game_over = False
you_win = False
player_jumping = False
jump_vel = 0.0
player_start_y = player.Y

#repeating loop
while True:
    framerate.tick(30)
    ticks = pygame.time.get_ticks()

    for event in pygame.event.get():
        if event.type == QUIT: sys.exit()
    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]: sys.exit()
    elif keys[K_SPACE]:
        if not player_jumping:
            player_jumping = True
            jump_vel = -8.0

    #update the arrow
    if not game_over:
        arrow.X -= arrow_vel
        if arrow.X < -40: reset_arrow()

    #did arrow hit player?
    if pygame.sprite.collide_rect(arrow, player):
        reset_arrow()
        player.X -= 10

    #did arrow hit dragon?
    if pygame.sprite.collide_rect(arrow, dragon):
        reset_arrow()
        dragon.X -= 10

    #did dragon eat the player?
    if pygame.sprite.collide_rect(player, dragon):
        game_over = True

    #did the dragon get defeated?
    if dragon.X < -100:
        you_win = True
        game_over = True

    #is the player jumping?
    if player_jumping:
        player.Y += jump_vel
        jump_vel += 0.5
        if player.Y > player_start_y:
            player_jumping = False
            player.Y = player_start_y
            jump_vel = 0.0

    #draw the background
    screen.blit(bg, (0,0))
```

```
    #update sprites
    if not game_over:
        group.update(ticks, 50)

    #draw sprites
    group.draw(screen)

    print_text(font, 350, 560, "Press SPACE to jump!")

    if game_over:
        print_text(font, 360, 100, "G A M E   O V E R")
        if you_win:
            print_text(font, 330, 130, "YOU BEAT THE DRAGON!")
        else:
            print_text(font, 330, 130, "THE DRAGON GOT YOU!")

    pygame.display.update()
```

Summary

Whew, sprite programming requires a lot of code just to get started, doesn't it? I feel like we just invented our own Python module just to get animation up and running. The good news is we now have a great new class called `MySprite` that can be modified and enhanced for any purpose from here on out. I'm sure we'll add new features to it in future chapters.

Challenges

1. **Modify the chapter game so that it keeps track of score every time the player successfully jumps over a flaming arrow without getting hit.**
2. **Modify the game further by making it possible to let the player make *high jumps* by holding down the Space key longer than usual.**
3. **Lastly, add a new feature to the `MySprite` class. Any new feature you want that makes it better!**

This page intentionally left blank

CHAPTER 8

Sprite Collision Detection: The Zombie Mob Game

We briefly touched upon the subject of collision detection in Chapter 7 when we needed to know when the flaming arrows were hitting the player and the dragon. In that chapter game, just one type of collision detection was used, between just one sprite and another (one-to-one). However, Pygame supports several types of collision detection techniques that we will be learning to use in this chapter. The subject of sprite groups will also become more important as you will see in the chapter example, called The Zombie Mob Game, which will use a large group of zombies versus the player for some fast-action gameplay. These are fairly advanced topics but all of the concepts hold each other up rather than stand on their own, so the code does get easier after a time.

In this exciting chapter you will learn how to:

- Check for collisions between two sprites
- Check for collisions between whole groups of sprites
- Create an awesome game called The Zombie Mob Game

Examining The Zombie Mob Game

The Zombie Mob Game, shown in Figure 8.1, is a fast-paced game in which the player has to run away from the zombies while collecting food in order to survive. This type of gameplay helps to demonstrate collision testing quite well because so many sprites are involved in the game. The gameplay will be improved even further in the next chapter when custom levels are designed for the game while learning about arrays and tuples.

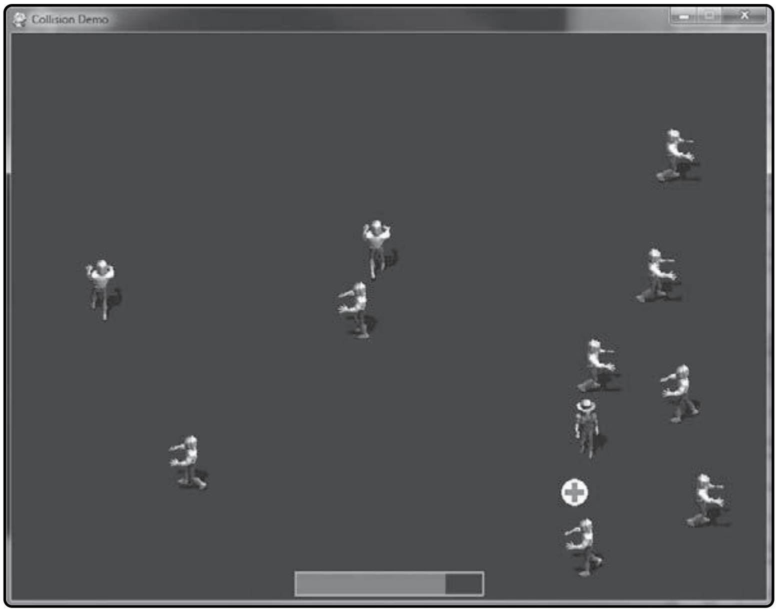


FIGURE 8.1

The Zombie Mob Game.

COLLISION DETECTION TECHNIQUES

Pygame supports several forms of collision detection that we can use for several different circumstances. Why do we need so many? Basically, for optimized code. Some forms of collision testing will involve just two sprites, while some test all of the sprites in an entire sprite group. It's even possible to test two groups against each other and get a list of all affected sprites in each group! This is a particularly interesting technique for a game like The Zombie Mob Game where there are item sprites to be picked up, a player sprite, and a large group of zombie sprites.

Rectangle Collision Between Two Sprites

One-on-one collision testing between just two sprites (rather than testing within a whole sprite group) is done with the `pygame.sprite.collide_rect()` function. Two parameters are passed, and each must be derived from `pygame.sprite.Sprite`. More specifically, *any* object can be passed as a parameter as long as it has a `Rect` property called `rect` available. In the function itself, `left.rect` and `right.rect` are used for the collision test, so if your first "sprite" (or any other object) has a `rect` property, then it will technically work, and the same goes for the second parameter called `right`. The function just returns a `bool` value (True or False) as a result of the collision test. This simple function will be your workhorse for custom sprite collision testing!

Using our custom `MySprite` class as a basis for examples, here is a simple one:

```
first = MySprite("battleship.png", 250, 120, 1)
second = MySprite("rowboat.png", 32, 16, 1)
result = pygame.sprite.collide_rect( first, second )
if result:
    print_text(font, 0, 0, "What were you thinking!?")
    sys.exit()
```

There's a variation of this function that we can also use for somewhat better results in some cases, depending on the sizes of the sprite images. The function is `pygame.sprite.collide_rect_ratio()`. The difference is, this function has an additional parameter—a float—where you can specify a percentage of the rectangle for the sprites to be used for collision. This is useful when there's a lot of empty space around the edges of a sprite image, in which case you'll want to make the rectangle smaller.

The syntax is a little strange, though, because the function actually creates an instance of a class with the reduction value, and then *that* result is passed the two sprite variable names as additional parameters.

```
pygame.sprite.collide_rect_ratio( 0.75 )( first, second )
```

Circle Collision Between Two Sprites

Circle collision is based on a radius value for each sprite. You can specify the radius yourself or let the `pygame.sprite.collide_circle()` function calculate the radius automatically. We might want to specify our own radius (as a new property of the sprite passed to this function) in order to fine-tune the collision results. If the `radius` property is not already there, then the function just calculates the radius based on the image size. The automatically created circle will not always produce very accurate collision results because the circle's radius completely encompasses the rectangle (that is, the diagonals rather than the width or height).

```
if pygame.sprite.collide_circle( first, second ):
    print_text(font, 0, 0, "Ha, I caught you!")
```

A variation of this function is also available with a float modifier parameter called `pygame.sprite.collide_circle_ratio()`.

```
pygame.sprite.collide_circle_ratio( 0.5 )( first, second )
```

Pixel-Perfect Masked Collision Between Two Sprites

The last collision testing function in `pygame.sprite` has the potential to be really awesome if used correctly! The function is `pygame.sprite.collide_mask()`, and receives two sprite variables as parameters, returning a `bool`.

```
if pygame.sprite.collide_mask( first, second ):
    print_text(font, 0, 0, "Argh, my pixels!")
```

Now, the awesome part is how this function works: if you supply a `mask` property in the `sprite` class, an image containing mask pixels for the sprite's collidable pixels, then the function will use it. Otherwise, the function will generate this mask on its own—and that's *very, very bad*. We definitely do not want a collision routine to mess with pixels every time it's called! Imagine if you have just 10 sprites using this function, all colliding with each other—that's 100 collision function calls, and *200* mask images being generated. So, this function has the potential to give really great collision results, but you simply *must* supply the mask image yourself!

To create a mask, look at the functions in the Surface module for reading and writing pixels. I'll give you a few hints: `Surface.lock()`, `Surface.unlock()`, `Surface.get_at()`, and `Surface.set_at()`. It's a lot of work, so unless your game really would benefit from this kind of precision, just use rectangular or circular collision instead!

HINT

Go ahead and give masked collision detection a try, but I don't recommend using it unless you have a slow-moving game where extreme precision is important. The other collision techniques work completely fine for 99 percent of the gameplay I've ever seen.

Rectangle Collision Between a Sprite and a Group

The first group collision function we're going to study now is `pygame.sprite.spritecollide()`. This function is surprisingly easy to use considering how much work it does. In a single function call, all of the sprites in a group are tested against another single sprite for collision, and a list of the collided sprites is returned as a result! The first parameter is the single sprite, while the second is the group. The third is a bool that really has great potential! Passing `True` here will cause all collided sprites in the group to be *removed*! That's a lot of hard work being done for us in a single function call. To manage the "damage," all sprites removed from the group are returned in the list!

```
collide_list = pygame.sprite.spritecollide( arrow, flock_of_birds, False)
```

Now, there is a variation of this function, which is probably not a big surprise given what we've seen already! The variation is `pygame.sprite.spritecollideany()`, and is a faster version of the function. Rather than returning all of the sprites in a list, it just returns a `bool` when a collision occurs with any sprite in the group. So, as soon as a collision occurs, it returns immediately.

```
if pygame.sprite.spritecollideany( arrow, flock_of_birds ):
    print_text(font, 0, 0, "Nice shot, you got one!")
```

The only problem is, you will have no way of knowing which sprite in the group was hit, but depending on gameplay that may not matter. Let me explain how this is useful. Imagine you have a maze-style game where all the walls of the maze are stored in a sprite group. Now, any time the player sprite has a collision with one of the walls in that group, it doesn't matter which wall, we just want to make the player stop moving. Presto, instant wall collision handling!

HINT

There's a mistake in the Pygame 1.9 docs related to the `spritecollideany()` function: the docs say the return value is a `bool`, but it is actually the `sprite` object in the group that collided with the other sprite passed to the function.

Rectangle Collision Between Two Groups

The last collision detection technique we'll look at is `pygame.sprite.groupcollide()`, which tests for collisions between two sprite groups. This is a potentially very intensive process and should not be used lightly if there are a very large number of sprites in either group being passed to it. The return from this function is a *dictionary* containing key-value pairs. Every sprite in the first group is added to the dictionary. Then, every sprite from group two that collides is added to the entry for group one in the dictionary. Some items from group one may be empty, while some might have many sprites from group two. Two additional `bool` parameters specify whether sprites should be removed from group one and two when a collision occurs.

```
hit_list = pygame.sprite.groupcollide( bombs, cities, True, False )
```

The Zombie Mob Game

We're going to use the information about collision detection now to make a game with a lot of sprites on the screen. The Zombie Mob Game pits the player against a mob of zombies. But, there's no weapon! This player character is a helpless civilian without weapons, and the goal is to avoid the zombies while collecting food in order to have energy to keep running away. The energy level drops every time the player moves, and if the energy reaches zero then the player won't be able to move any more and the zombies will have their favorite (and only) food for dinner—your brains.

Creating Your Own Module

Besides making the game using collision detection techniques, we're going to explore modular programming at this point too because our library of code is getting kind of repetitive. We have the `MySprite` class, the `print_text()` function, and as you may recall, the useful `Point` class introduced back in Chapter 6—these will be used frequently. So, we'll put them in a separate source code file for reuse. Python make this really easy to do, too! Just put your code in another file with a .py extension, and call it whatever you want. Then, in the program code where you want to use that helper code, add an import statement! I'm going to call the helper library file `MyLibrary.py`. So, in the game file we'll add this line:

```
import MyLibrary
```

Well, in a manner of speaking. If you do it that way, then you have to add `MyLibrary.` (with a period) in front of every class name and function in `MyLibrary.py` to use it. Not a big deal, but I want the code to pretty much remain as it has in past chapters. So, we'll use a variation of import that includes everything in the file into Python's global namespace:

```
from MyLibrary import *
```

One more thing: Any time you need to reference something in `MyLibrary`, you'll have to pass it as a parameter or create a local reference to the object. The `screen` variable, for example, is used for drawing. So, instead of passing it to every function that needs it, we can just call `pygame.display.get_surface()` to retrieve the existing surface. The `print_text()` function needs this line added, for example.

```
screen = pygame.display.get_surface()
```

Below is the source code for the `MyLibrary.py` file. Now, just go ahead and add any new functions or classes to this file and then copy the file to the folder where any of your Python/Pygame games are located, so that you can use it. Just note that we will not be including these classes and functions in any future examples, so take note of where they went!

```
# MyLibrary.py
import sys, time, random, math, pygame
from pygame.locals import *

# prints text using the supplied font
def print_text(font, x, y, text, color=(255,255,255)):
    imgText = font.render(text, True, color)
    screen = pygame.display.get_surface()
    screen.blit(imgText, (x,y))
```

```
# MySprite class extends pygame.sprite.Sprite
class MySprite(pygame.sprite.Sprite):

    def __init__(self):
        pygame.sprite.Sprite.__init__(self) #extend the base Sprite class
        self.master_image = None
        self.frame = 0
        self.old_frame = -1
        self.frame_width = 1
        self.frame_height = 1
        self.first_frame = 0
        self.last_frame = 0
        self.columns = 1
        self.last_time = 0

    #X property
    def _getx(self): return self.rect.x
    def _setx(self,value): self.rect.x = value
    X = property(_getx,_setx)

    #Y property
    def _gety(self): return self.rect.y
    def _sety(self,value): self.rect.y = value
    Y = property(_gety,_sety)

    #position property
    def _getpos(self): return self.rect.topleft
    def _setpos(self,pos): self.rect.topleft = pos
    position = property(_getpos,_setpos)

    def load(self, filename, width, height, columns):
        self.master_image = pygame.image.load(filename).convert_alpha()
        self.frame_width = width
        self.frame_height = height
        self.rect = Rect(0,0,width,height)
        self.columns = columns
        #try to auto-calculate total frames
        rect = self.master_image.get_rect()
        self.last_frame = (rect.width // width) * (rect.height // height) - 1

    def update(self, current_time, rate=30):
        #update animation frame number
        if current_time > self.last_time + rate:
            self.frame += 1
            if self.frame > self.last_frame:
                self.frame = self.first_frame
            self.last_time = current_time

        #build current frame only if it changed
        if self.frame != self.old_frame:
            frame_x = (self.frame % self.columns) * self.frame_width
            frame_y = (self.frame // self.columns) * self.frame_height
            rect = Rect(frame_x, frame_y, self.frame_width, self.frame_height)
            self.image = self.master_image.subsurface(rect)
            self.old_frame = self.frame

    def __str__(self):
        return str(self.frame) + "," + str(self.first_frame) + \
               "," + str(self.last_frame) + "," + str(self.frame_width) + \
               "," + str(self.frame_height) + "," + str(self.columns) + \
               "," + str(self.rect)

#Point class
class Point(object):
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    #X property
    def getx(self): return self.__x
    def setx(self, x): self.__x = x
    x = property(getx, setx)

    #Y property
    def gety(self): return self.__y
    def sety(self, y): self.__y = y
    y = property(gety, sety)

    def __str__(self):
        return "{X:" + "{:.0f}".format(self.__x) + \
            ",Y:" + "{:.0f}".format(self.__y) + "}"
```

Advanced Directional Animation

Our Zombie Mob Game uses some artwork to make it look really cool. I was going to make the zombies just as green circles and the player as a white circle, but that would not make it onto the cover of any game development magazines or the front page of any web logs, so this game will be using good artwork! The player character's artwork is shown in Figure 8.2.

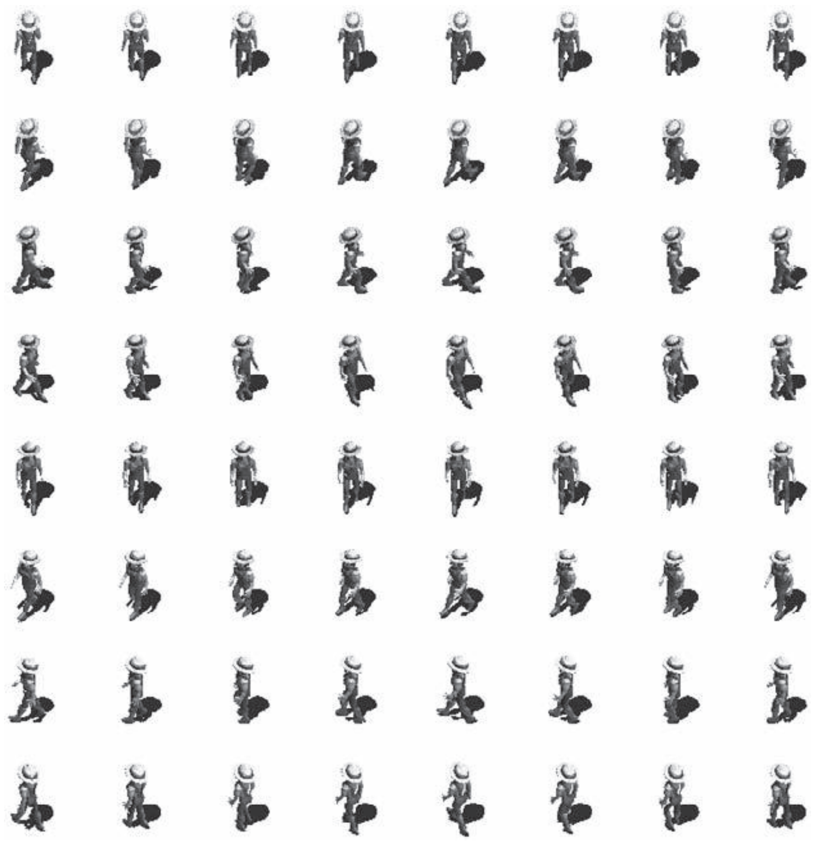


FIGURE 8.2

Sprite sheet of the animated walking player character.

Note the specifications of this sprite sheet, since we'll have to know this information for our source code. There are eight columns across, and eight rows, so that's 64 total frames. You can't tell from the figure, but by opening the bitmap file in a graphic editor to view it, you will note that the file dimensions are 768 × 768. That breaks down to a frame size of 96 × 96 pixels. But, we don't really need all of these frames of animation. This is a *great* sprite sheet for a game that could use eight directions of movement: north, south, east, west, and all four diagonals. Our zombie game will just be using the four primary directions—but going to eight could be an upgrade to the game if you want to tackle it! Figure 8.3 shows the sprite sheet for the zombie. All of the zombie sprites will share this one bitmap file.

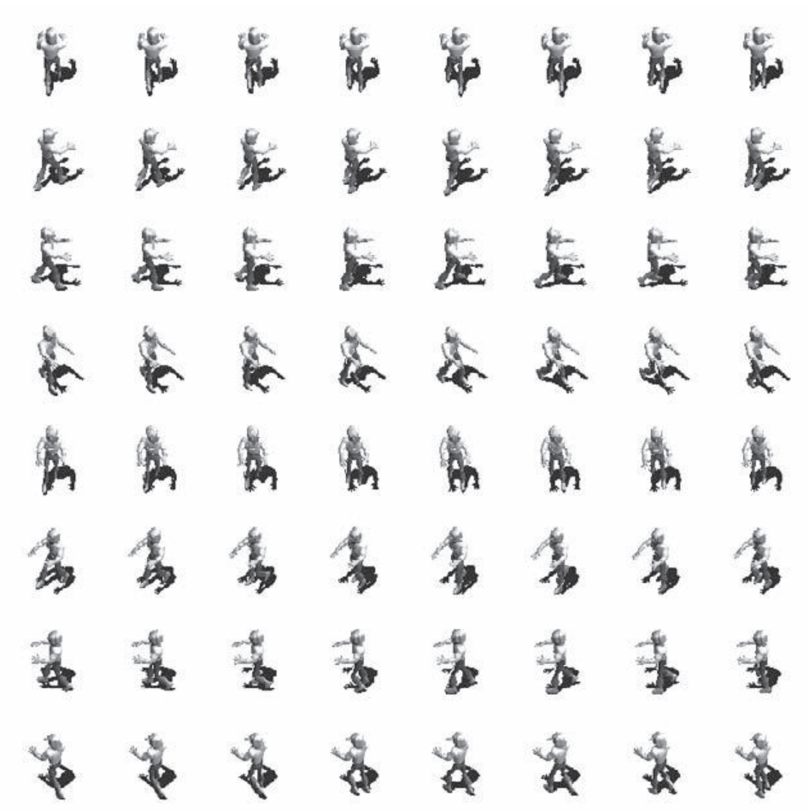


Figure 8.3
Sprite sheet of the animated walking zombie.

To make these sprites move in the four directions when there are eight animation sequences, we have to manually control the animation frame ranges. Table 8.1 shows the specifications for the animations. Since both the player and zombie sprite sheets have the same dimensions, this applies to both. While studying the table's figures, use the figures showing the two sprite sheets as a reference. I've found it actually helps to *count* each frame in each row. Once you have counted the first few rows, you should notice a pattern emerge—based on the number of columns (eight). Since this is a pattern, we can use it to automatically calculate the range for each direction.

TABLE 8.1 SPRITE SHEET DIMENSIONS

Row	Description	Start Frame	End Frame
0	north	0	7
1	--	8	15
2	east	16	23
3	--	24	31
4	south	32	39
5	--	40	47
6	west	48	55
7	--	56	63

The direction value will be added to the `sprite` class as a new property. While we're at it, we will also need a velocity property added to `MySprite` so that the sprite can be moved based on its direction. So, yes, we will need to open up `MyLibrary` and make an addition to the `MySprite` class. That's what it's there for, so don't be afraid to modify it! Let's do that right now while we're thinking about it:

```
class MySprite(pygame.sprite.Sprite):
    def __init__(self):
        pygame.sprite.Sprite.__init__(self) #extend the base Sprite class
        self.master_image = None
        self.frame = 0
        self.old_frame = -1
        self.frame_width = 1
        self.frame_height = 1
        self.first_frame = 0
        self.last_frame = 0
        self.columns = 1
        self.last_time = 0
        self.direction = 0
        self.velocity = Point(0,0)
```

That's all we have to do to add it as a global property. Oh, it's not a *real* Python class property, but it will work fine this way, like the others. The only time I create a real *property* with a `get()` and `set()` pair (which are called the accessor/mutator methods, by the way), is when either one has to do some logic, as was the case with the X and Y properties.

Now, we have a `MySprite.direction` property available, so what we want to do is set the direction based on user input. When the player presses the Up key, we will set the direction to 0 (north). Likewise, we'll do the same for the Right key (2, east), the Down key (4, south), and the Left key (6, west). The code below takes into account both the arrow keys and W-A-S-D keys commonly used for movement.

```
if keys[K_UP] or keys[K_w]:      player.direction = 0
elif keys[K_RIGHT] or keys[K_d]: player.direction = 2
elif keys[K_DOWN] or keys[K_s]:  player.direction = 4
elif keys[K_LEFT] or keys[K_a]:  player.direction = 6
```

The direction will then determine the frame range that will be used for animation. So, if you press the Up key, the range 0 to 7 will be used, and so on according to Table 8.1. The great thing about the `direction` property is that we can use it in a simple calculation to set the range. No `if` statement is needed!

```
player.first_frame = player.direction * player.columns
player.last_frame = player.first_frame + 8
```

We'll want to be sure this code comes before `player.update()`, which is where animation is updated.

Colliding with Zombies

Our collision code in the game will involve two stages. First, we'll use `pygame.sprite.spritecollideany()` to see if the player sprite touches any zombie sprite. If that comes back with a hit, then we'll do a second collision test using `pygame.sprite.collide_rect_ratio()` and reduce the collision boxes by 50 percent for a more accurate result—which leads to better gameplay. The reason this second stage is needed is because the frames in the sprite sheets are quite large compared to the actual image pixels in each frame, so the collision needs to be *tightened up* a bit for better gameplay. A scratch sprite object called `attacker` is used to track when the player has been hit by a zombie. After the two-stage collision checks pass, then the player loses health, and the zombie is pushed back a little ways to give the player room to escape. Figure 8.4 shows the player about to be attacked!

TRAP

Be careful when writing code that causes something to reverse direction (or any other state), because if it isn't also moved out of that position (or state), then it will keep flip-flopping back and forth and seem to "wig out" on the screen. At worst, this can actually lock up the game.

```
#check for collision with zombies
attacker = None
attacker = pygame.sprite.spritecollideany(player, zombie_group)
if attacker != None:
    #we got a hit, now do a more precise check
    if pygame.sprite.collide_rect_ratio(0.5)(player,attacker):
        player_health -= 10
        if attacker.X < player.X:   attacker.X -= 10
        elif attacker.X > player.X: attacker.X += 10
    else:
        attacker = None
```

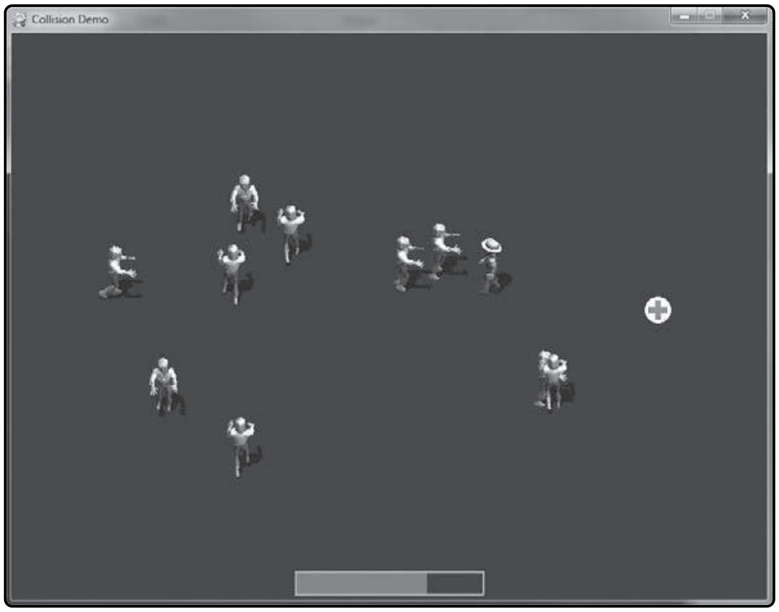


FIGURE 8.4

The player is getting attacked by the zombies!

Getting Health

The health sprite is a little red cross on a white circle that the player can pick up to gain +30 health (of course you may change this if you want to make the game harder or easier!). The code to let the player pick up the `health` sprite looks like this. When the `health` sprite is picked up, then the player receives bonus health and it is moved to a new random location on the screen. Figure 8.5 shows the `health` sprite and the player needs it badly in this case!

```
#check for collision with health
if pygame.sprite.collide_rect_ratio(0.5)(player,health):
    player_health += 30
    if player_health > 100: player_health = 100
    health.X = random.randint(0,700)
    health.Y = random.randint(0,500)
```

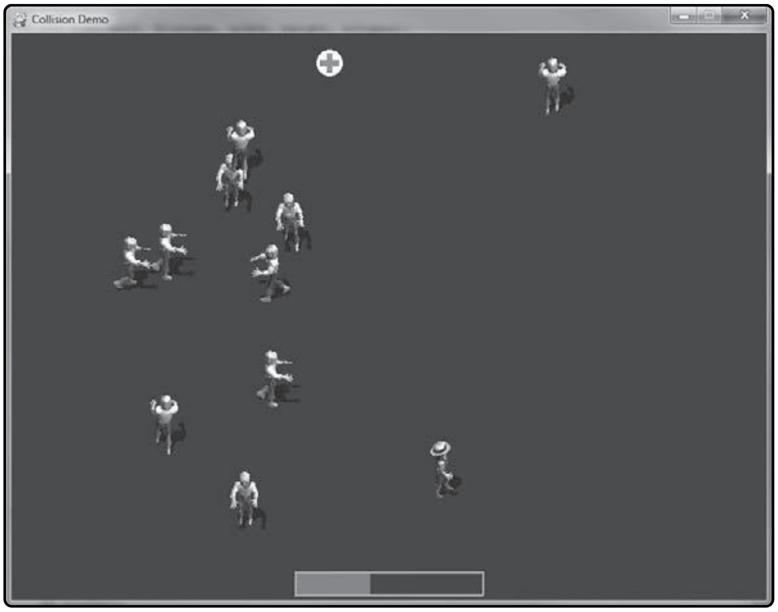


Figure 8.5

That health pickup is *really* far away!

TRICK

The player can move just *slightly* faster than the zombies, which makes the game fun. If the player moves at the same or slower speed than the zombies, then the gameplay would be frustrating. Always give your player the edge over the bad guys so they'll keep coming back to play. Frustrating the player is a sure-fire way to make them quit playing.

If the player gets attacked by the zombies too much and can't get to the health powerup, then eventually the health bar runs out and the player dies. This marks the end of the game. There is no way to reset the game other than by running it again.

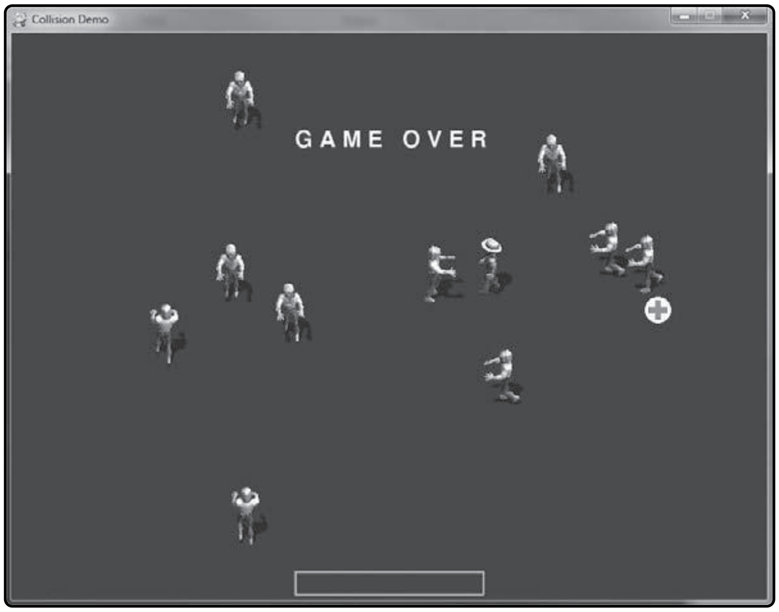


Figure 8.6

Oh no, the player has died!

Game Source Code

Here, finally, is the complete source code for The Zombie Mob Game. It's a rather short listing considering how much gameplay this game is packing! Thanks to the `MyLibrary.py` file, we've managed to store away the reusable code and tighten up our game's main code listing.

```
# Zombie Mob Game
# Chapter 8
import itertools, sys, time, random, math, pygame
from pygame.locals import *
from MyLibrary import *

def calc_velocity(direction, vel=1.0):
    velocity = Point(0,0)
    if direction == 0: #north
        velocity.y = -vel
    elif direction == 2: #east
        velocity.x = vel
    elif direction == 4: #south
        velocity.y = vel
    elif direction == 6: #west
```

```
        velocity.x = -vel
    return velocity

def reverse_direction(sprite):
    if sprite.direction == 0:
        sprite.direction = 4
    elif sprite.direction == 2:
        sprite.direction = 6
    elif sprite.direction == 4:
        sprite.direction = 0
    elif sprite.direction == 6:
        sprite.direction = 2

#main program begins
pygame.init()
screen = pygame.display.set_mode((800,600))
pygame.display.set_caption("Collision Demo")
font = pygame.font.Font(None, 36)
timer = pygame.time.Clock()

#create sprite groups
player_group = pygame.sprite.Group()
zombie_group = pygame.sprite.Group()
health_group = pygame.sprite.Group()

#create the player sprite
player = MySprite()
player.load("farmer walk.png", 96, 96, 8)
player.position = 80, 80
player.direction = 4
player_group.add(player)

#create the zombie sprite
zombie_image = pygame.image.load("zombie walk.png").convert_alpha()
for n in range(0, 10):
    zombie = MySprite()
    zombie.load("zombie walk.png", 96, 96, 8)
    zombie.position = random.randint(0,700), random.randint(0,500)
    zombie.direction = random.randint(0,3) * 2
    zombie_group.add(zombie)

#create heath sprite
health = MySprite()
health.load("health.png", 32, 32, 1)
health.position = 400,300
health_group.add(health)

game_over = False
player_moving = False
player_health = 100

#repeating loop
while True:
    timer.tick(30)
    ticks = pygame.time.get_ticks()

    for event in pygame.event.get():
        if event.type == QUIT: sys.exit()
    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]: sys.exit()
    elif keys[K_UP] or keys[K_w]:
        player.direction = 0
        player_moving = True
    elif keys[K_RIGHT] or keys[K_d]:
        player.direction = 2
        player_moving = True
    elif keys[K_DOWN] or keys[K_s]:
        player.direction = 4
        player_moving = True
    elif keys[K_LEFT] or keys[K_a]:
        player.direction = 6
        player_moving = True
    else:
        player_moving = False

    #these things should not happen when the game is over
```

```
if not game_over:
    #set animation frames based on player's direction
    player.first_frame = player.direction * player.columns
    player.last_frame = player.first_frame + player.columns-1
    if player.frame < player.first_frame:
        player.frame = player.first_frame

    if not player_moving:
        #stop animating when player is not pressing a key
        player.frame = player.first_frame = player.last_frame
    else:
        #move player in direction
        player.velocity = calc_velocity(player.direction, 1.5)
        player.velocity.x *= 1.5
        player.velocity.y *= 1.5

    #update player sprite
    player_group.update(ticks, 50)

    #manually move the player
    if player_moving:
        player.X += player.velocity.x
        player.Y += player.velocity.y
        if player.X < 0: player.X = 0
        elif player.X > 700: player.X = 700
        if player.Y < 0: player.Y = 0
        elif player.Y > 500: player.Y = 500

    #update zombie sprites
    zombie_group.update(ticks, 50)

    #manually iterate through all the zombies
    for z in zombie_group:
        #set the zombie's animation range
        z.first_frame = z.direction * z.columns
        z.last_frame = z.first_frame + z.columns-1
        if z.frame < z.first_frame:
            z.frame = z.first_frame
        z.velocity = calc_velocity(z.direction)

        #keep the zombie on the screen
        z.X += z.velocity.x
        z.Y += z.velocity.y
        if z.X < 0 or z.X > 700 or z.Y < 0 or z.Y > 500:
            reverse_direction(z)


    #check for collision with zombies
    attacker = None
    attacker = pygame.sprite.spritecollideany(player, zombie_group)
    if attacker != None:
        #we got a hit, now do a more precise check
        if pygame.sprite.collide_rect_ratio(0.5)(player,attacker):
            player_health -= 10
            if attacker.X < player.X:
                attacker.X -= 10
            elif attacker.X > player.X:
                attacker.X += 10
        else:
            attacker = None

    #update the health drop
    health_group.update(ticks, 50)

    #check for collision with health
    if pygame.sprite.collide_rect_ratio(0.5)(player,health):
        player_health += 30
        if player_health > 100: player_health = 100
        health.X = random.randint(0,700)
        health.Y = random.randint(0,500)

#is player dead?
if player_health <= 0:
    game_over = True

#clear the screen
```

```
screen.fill((50,50,100))

#draw sprites
health_group.draw(screen)
zombie_group.draw(screen)
player_group.draw(screen)

#draw energy bar
pygame.draw.rect(screen, (50,150,50,180), Rect(300,570,player_health*2,25))
pygame.draw.rect(screen, (100,200,100,180), Rect(300,570,200,25), 2)

if game_over:
    print_text(font, 300, 100, "G A M E   O V E R")

pygame.display.update()
```

IN THE REAL WORLD START

If you *really* love zombies, then go to the source—George A. Romero, who created the zombie genre on film starting with the original classic, *Night of the Living Dead*. This film has led to modern marvels such as the *Resident Evil* series starring Milla Jovovich, TV shows like *The Walking Dead*, and even remakes of Romero's films including the new renditions of *The Crazies*, *Dawn of the Dead*, and *Day of the Dead*. This is must-watch inspirational material for anyone on the fast-track to make a zombie game!

SUMMARY

That concludes our experiments in surviving the apocalypse. Sprite collision detection was also somewhat important in this chapter as well. The Zombie Mob Game was a pretty good example of several types of collision in practice. As the game code demonstrated, the *response* to collision events is extremely important.

Challenges

1. **Modify the game so that a new zombie is added to the `zombie_group` group every 10 seconds or so, using the `timer` variable. This will ramp up the difficulty and give the player a greater challenge (and also make it impossible to survive at a certain point, like any good arcade game!).**
2. **Modify the game so that there is more than one health pickup sprite.**
3. **Modify the zombie collision code so that the zombies collide with each other, using code similar to the collision response with the player.**

CHAPTER 9

Arrays, Lists, and Tuples: The Block Breaker Game

This chapter explores the rather mysterious subject of arrays and the related subject of *tuples* that goes along with it. The two are essentially the same thing: an array when used simply like an array, and a tuple when treated like a container object with properties and methods. A `list` is a Python class that we will also learn to use. We will use this new knowledge to create a game with pre-defined levels.

In this chapter, you will learn to:

- Define and use arrays and lists
- Use a tuple as a constant array of data
- Create a data-driven game

Examining The Block Breaker Game

The Block Breaker Game uses the concepts presented in this chapter. As a data-driven game, it will be possible to make changes to the game level definitions which will change the appearance of those levels and affect the gameplay without changing any other lines of code. This will be a good demonstration of lists and tuples.

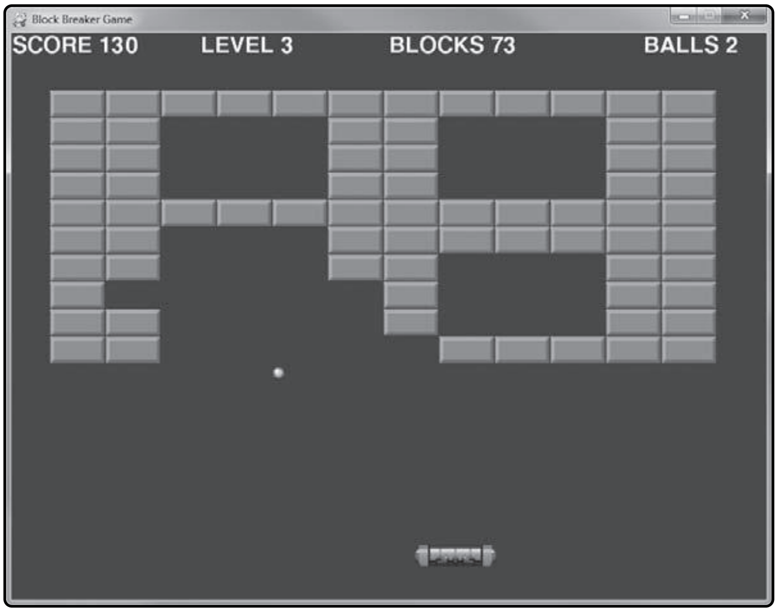


Figure 9.1

The Block Breaker Game.

Arrays and Lists

Since arrays are just simplified lists, we can combine these two in a single section and cover lists to encompass both. A list is a container of data—any data you want to store using normal Python variables. A list can also contain *objects* based on your own classes, like `MySprite`. In *fact*, a sprite group, such as `pygame.sprite.Group`, is just a list! So, you should already feel somewhat familiar with the subject after having used lists already. Lists are considered *mutable* because the elements in a list can be changed, and the list can be modified in various ways by adding, removing, searching, and sorting.

Lists with One Dimension

A list is created by either defining the elements all at once, or by adding elements at a later time. For example:

```
ages = [16, 91, 29, 38, 14, 22]
print(ages)
[16, 91, 29, 38, 14, 22]
```

Lists can contain other data besides integers, such as strings.

```
names = ["john","jane","dave","robert","andrea","susan"]
print(names)
['john', 'jane', 'dave', 'robert', 'andrea', 'susan']
```

Changing One Element

We can get the data from any element in the list by index number. The element can also be changed by referencing the index number. Here, we'll change the value in index 1, then reset it afterward.

```
ages[1] = 1000
print(ages[1])
1000
ages[1] = 91
```

Adding One Element

New items can be added to the list with the `append()` method:

```
ages.append(100)
print(ages)
[16, 91, 29, 38, 14, 22, 100]
```

An element can be inserted into the middle of the list with the `insert()` method, which accepts an index position and a value.

```
ages.insert(1, 50)
print(ages)
ages.insert(1, 60)
print(ages)
[16, 50, 91, 29, 38, 14, 22, 100, 20, 20, 20]
[16, 60, 50, 91, 29, 38, 14, 22, 100, 20, 20, 20]
```

Counting Elements

If there are duplicate elements in the list, they can be counted using the `count()` method.

```
ages.append(20)
ages.append(20)
ages.append(20)
print(ages)
print(ages.count(20))
[16, 91, 29, 38, 14, 22, 100, 20, 20, 20]
3
```

Searching for Elements

A list can be searched for the first occurrence of a specific element with the `index()` method. Note that a list is zero-based, so the first item is in index position zero, not one. The first occurrence of the value 20 in the list at this point is index 7 (the 8th element).

```
print(ages.index(20))
7
```

Removing Elements

An element of the list can be removed with the `remove()` method. The first occurrence of the value passed will be the one removed, and just one, not all. In the code below, note that the first occurrence of `20` that was recently added is removed.

```
ages.remove(20)
print(ages)
[16, 60, 50, 91, 29, 38, 14, 22, 100, 20, 20]
```

Reversing a List

The entire list can be reversed with the `reverse()` method. This has the effect of changing every element in the list. The following code shows the result when the elements in our sample list have been reversed. They are then returned to the original ordering with a duplicate call to `reverse()`.

```
ages.reverse()
print(ages)
[20, 20, 100, 22, 14, 38, 29, 91, 50, 60, 16]
ages.reverse()
```

Sorting a List

The elements of a list can be sorted with the `sort()` method. As the sample code below demonstrates, the sort order can be reversed with a call to `reverse()`, so a descending sort is not needed.

```
ages.sort()
print(ages)
ages.reverse()
print(ages)
[14, 16, 20, 20, 22, 29, 38, 50, 60, 91, 100]
[100, 91, 60, 50, 38, 29, 22, 20, 20, 16, 14]
```

Creating a Stack-like List

A stack is a list with a first-in, last-out (FILO) mechanism for managing elements, where the last item added is the first item removed. A method called `pop()` makes it a little easier to use a list like a stack by removing the last item on the list. The usual parlance for stack programming is that elements are "pushed" onto the stack, not just added or appended, but we can use `append()` for the same purpose. A stack is a great tool for managing short-term memory, and it is the technique used by compilers and interpreters to read parameters passed to a function.

```
stack = []
for i in range(10):
    stack.append(i)
print(stack)
stack.append(10)
print(stack)
n = stack.pop()
m = stack.pop()
print(stack)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Creating a Queue-like List

A queue is similar to a stack in functionality, but it uses a first-in, first-out (FIFO) mechanism for managing elements, where the first item added is the first item removed. Python already has a queue module that can be used for this purpose, so we're only simulating one with a list for illustration.

```
queue = []
for l in range(10):
    queue.append(l)
print(queue)
queue.append(50)
queue.append(60)
queue.append(70)
print(queue)
n = queue[0]
queue.remove(n)
print(queue)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 50, 60, 70]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 50, 60, 70]
```

Lists with More Dimensions

A list can also contain lists, which is called a multi-dimensional list. A two-dimensional list might be called a grid, since the data will look like a spreadsheet. This is a common technique for storing game level data. Working with n-dimensional lists in Python can be tricky until you learn the syntax. Here is a two-dimensional list:

```
grid = [[1,2,3],[4,5,6],[7,8,9]]
print(grid)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Note the syntax for creating the list values, with brackets separated by commas. The first dimension is represented by the outer brackets. If you remove the second dimension, the list becomes:

```
grid = []
```

which is what we would expect. Adding the second dimension is a matter of syntax. To make the second dimension easier to visualize, we can use a more visually appealing form of definition:

```
grid = [
    [1,2,3],
    [4,5,6],
    [7,8,9]]
```

Python doesn't "see" an n-dimensional list in this manner, so it is helpful only to the programmer.

Changing One Element

To change a single element in a two-dimensional list such as this one, we have to use an index within brackets for the syntax. For example, the following prints out the first "element" of the list's first dimension followed by the size of the element:

```
print(grid[0])
print(len(grid[0]))
[1, 2, 3]
3
```

Here is another way to display the elements in the list contained in `grid[0]`:

```
for n in grid[0]: print(n)
1
2
3
```

Understanding that an element can be another list helps when working with lists such as this. Just add a second pair of brackets with an index value to access the values in that inner list.

```
grid[0][0] = 100
grid[0][1] = 200
grid[0][2] = 300
print(grid[0])
[100, 200, 300]
```

Changing Many Elements

The fastest way to fill a list of lists (another term for a two-dimensional list) with data is with a `for` loop. The following syntax can be used to fill a new list of lists with a single value:

```
grid = [
    [10 for col in range(10)]
        for row in range(10)]
for row in grid: print(row)
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

We can also define a whole list with a unique value for each element position. Here is a list definition for a game level with 12 columns and 10 rows:

```
level = [
    1,1,1,1,1,1,1,1,1,1,1,1,
    2,2,2,2,2,2,2,2,2,2,2,2,
    3,3,3,3,3,3,3,3,3,3,3,3,
    1,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,0,0,1,1,1,1,1,
    1,1,1,1,1,0,0,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,
    3,3,3,3,3,3,3,3,3,3,3,3,
    2,2,2,2,2,2,2,2,2,2,2,2,
    1,1,1,1,1,1,1,1,1,1,1,1]
```

When printing out the list, there will be no formatting of lines because this is just one long definition of data for an element that just so happens to be the level data for a game. If the elements must be lined up, then a little bit of finagling of the index has to be done with a pair of `for` loops. It helps to remember that the first dimension represents "Y" while the second or inner dimension represents "X," if you want to think of the data in the list in terms of X and Y coordinates. In that case, each row will be processed first, and the elements inside each row (the columns) will be processed in order per row.

Here's one way to do it if you wish to have some control over how the values are printed out:

```
for row in range(10):
    s = ""
    for col in range(12):
        s += str(level[row*10+col]) + " "
    print(s)
1 1 1 1 1 1 1 1 1 1 1 1
1 1 2 2 2 2 2 2 2 2 2 2
2 2 2 2 3 3 3 3 3 3 3 3
3 3 3 3 3 3 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 1 1 1 1 1 1 1
1 1 1 1 1 0 0 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 3 3 3 3 3 3 3 3
3 3 3 3 3 3 2 2 2 2 2 2
```

This rather complex formatting of the elements is necessary because this is a single-dimension list disguised as a two-dimensional one! The tip-off is the index calculation: `row * 10 + col`, which is based on the formula for converting an index from two dimensions into one dimension:

```
index = (row #) x columns + (column #)
```

To simplify the code and eliminate this calculation, we can just define the level data as a two-dimensional list in the definition itself:

```
level = [
    [1,1,1,1,1,1,1,1,1,1,1,1],
    [2,2,2,2,2,2,2,2,2,2,2,2],
    [3,3,3,3,3,3,3,3,3,3,3,3],
    [1,1,1,1,1,1,1,1,1,1,1,1],
    [1,1,1,1,1,0,0,1,1,1,1,1],
    [1,1,1,1,1,0,0,1,1,1,1,1],
    [1,1,1,1,1,1,1,1,1,1,1,1],
    [3,3,3,3,3,3,3,3,3,3,3,3],
    [2,2,2,2,2,2,2,2,2,2,2,2],
    [1,1,1,1,1,1,1,1,1,1,1,1]]
```

To print out (or just access the list in general), we have only to treat each row as a list within the list using a simple `for` loop:

```
for row in level: print(row)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Tuples

A tuple is similar to a list, but it is read-only, meaning the items cannot be changed once they have been initialized in code—which makes tuples *immutable*. The elements in a tuple are enclosed in parentheses rather than brackets to denote the difference. Once defined, a tuple can only be replaced. Why use a tuple instead of a list? The main advantage to a tuple is that it is *faster* than a list. If you will not be changing the data, then use a tuple for better performance. But if you need to change data, then use a list.

Packing a Tuple

The process of creating a tuple is called *packing*. Tuples are used often to pass complex data to and from functions and class methods. A tuple's data can only be created once and then it is immutable. In the short example below, a tuple is created containing the values 1 through 5. Then variables a, b, c, d, and e are set to the value of each respective element in the tuple. If the code looks familiar, that's because we've been using tuples fairly regularly up to this point without formally recognizing them!

```
tuple1 = (1,2,3,4,5)
print(tuple1)
(1, 2, 3, 4, 5)
```

Unpacking a Tuple

Note that the parentheses are optional when working with a tuple, as the following code illustrates. The process of reading data out of a tuple is called "unpacking."

```
a,b,c,d,e = tuple1
print(a,b,c,d,e)
1 2 3 4 5
```

More complex tuples can be created with code:

```
data = (100 for n in range(10))
for n in data: print(n)
100
100
100
100
100
100
100
100
100
100
```

String data can also be stored in a tuple:

```
names = ("john","jane","dave","robert","andrea","susan")
print(names)
('john', 'jane', 'dave', 'robert', 'andrea', 'susan')
```

Searching for Elements

Some methods commonly found when working with a list are also available to a tuple, but only those methods that retrieve data, not change it. Trying to change data in a tuple generates a run-time error in the Python interpreter.

```
print(names.index("dave"))
2
```

Elements in a tuple can be searched using range sequence operators such as `in`:

```
print("jane" in names)
True
print("bob" in names)
False
```

Counting Elements

We can have Python return the number of elements of a specific value stored in a tuple using the `count()` method:

```
print(names.count("susan"))
1
```

We can also get the length of all elements in a tuple with the `len()` function:

```
print(len(names))
6
```

Tuples as Constant Arrays

Tuples work exceptionally well as a constant array container due to its ability to search and return data quickly. The syntax will be similar to that for lists, but using parentheses rather than brackets. Here is a 2D tuple containing level data for a game:

```
level = (
    (1,1,1,1,1,1,1,1,1,1,1,1),
    (2,2,2,2,2,2,2,2,2,2,2,2),
    (3,3,3,3,3,3,3,3,3,3,3,3),
    (1,1,1,1,1,1,1,1,1,1,1,1),
    (1,1,1,1,1,0,0,1,1,1,1,1),
    (1,1,1,1,1,0,0,1,1,1,1,1),
    (1,1,1,1,1,1,1,1,1,1,1,1),
    (3,3,3,3,3,3,3,3,3,3,3,3),
    (2,2,2,2,2,2,2,2,2,2,2,2),
    (1,1,1,1,1,1,1,1,1,1,1,1))
```

Accessing the data in a 2D tuple can be done with the same code used to access a list, using a `for` loop or by index.

```
for row in level: print(row)
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
(3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
(1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1)
(1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1)
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
(3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)
(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

The Block Breaker Game

We're going to put our new knowledge of lists and tuples to work in a game called Block Breaker. This is a traditional ball-and-paddle game where the goal is to clear all of the blocks from the playing field while keeping the ball from getting past the paddle. This is essentially "Ping Pong"–style gameplay for one player. We're going to build the game section by section and explain it as we go rather than showing all of the source code at once. Let's begin with the imports:

```
# Block Breaker Game
# Chapter 9
import sys, time, random, math, pygame
from pygame.locals import *
from MyLibrary import *
```

Note that `MyLibrary.py` is needed. We'll be making a few changes to the library's source code here in a bit!

Block Breaker Levels

There are three levels in the game now, but you are welcome to add new levels to the game or edit the ones already defined. The game code uses `len(levels)` when changing the game level, so you can add as many new levels as you want to the levels tuple without having to change anything in the source code dealing with level changing limits. Figure 9.2 shows the block sprite sheet image.

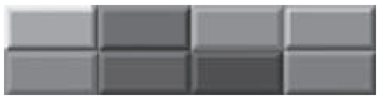


FIGURE 9.2

The blocks all share this image, which contains 8 frames.

Level 1

Figure 9.3 shows the first level of the game. It is not important to have 1's defined in the level 1 data; that was just done to help illustrate the level this is referring to. Although the block image used for the game is treated like an animated sprite, we could have just used a simple white block and colored it when drawing it with any color value.

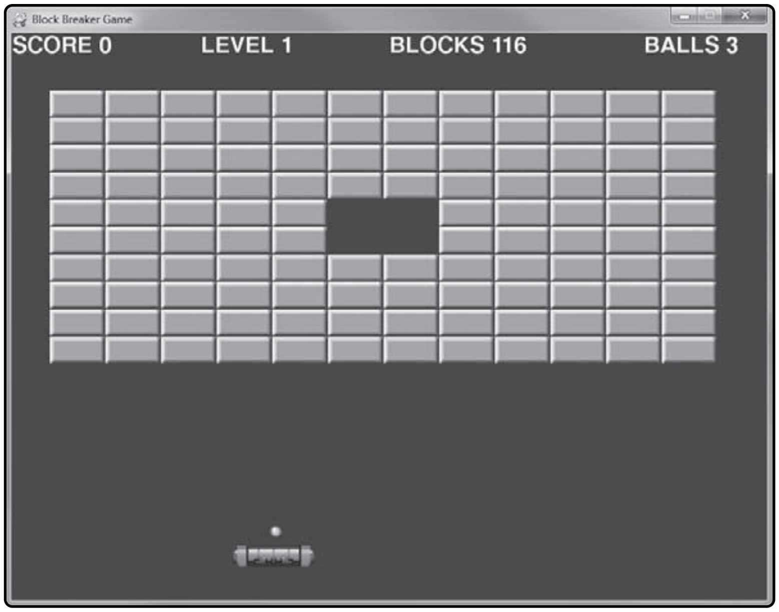


FIGURE 9.3

Level 1 of the game.

```
levels = (
(1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,0,0,1,1,1,1,1,
 1,1,1,1,1,0,0,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1),
```

Level 2

Figure 9.4 shows the second level of the game. Like the level 1 data, the 2's found herein are only illustrative. You may change them to any value from 0 to 7, as there are 8 different blocks.

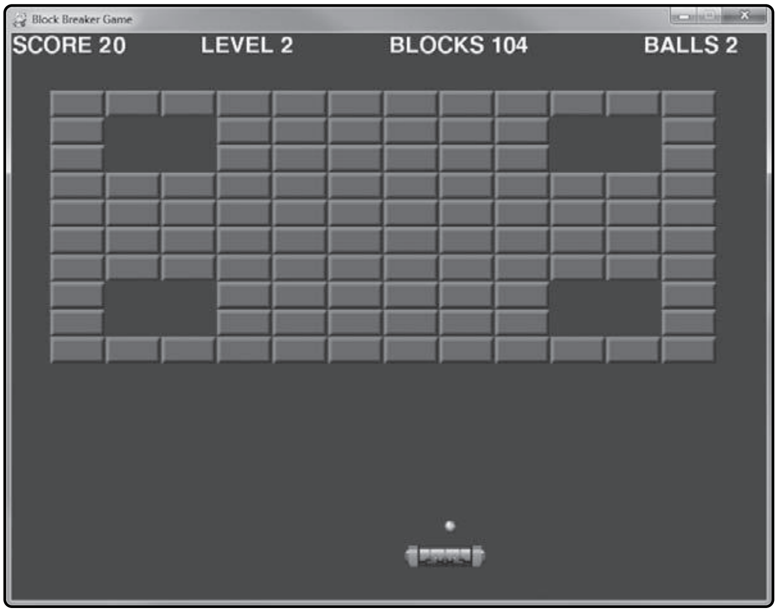


FIGURE 9.4

Level 2.

```
(2,2,2,2,2,2,2,2,2,2,2,2,
 2,0,0,2,2,2,2,2,2,0,0,2,
 2,0,0,2,2,2,2,2,2,0,0,2,
 2,2,2,2,2,2,2,2,2,2,2,2,
 2,2,2,2,2,2,2,2,2,2,2,2,
 2,2,2,2,2,2,2,2,2,2,2,2,
 2,2,2,2,2,2,2,2,2,2,2,2,
 2,0,0,2,2,2,2,2,2,0,0,2,
 2,0,0,2,2,2,2,2,2,0,0,2,
 2,2,2,2,2,2,2,2,2,2,2,2),
```

Level 3

Figure 9.5 shows the third and final level of the game.

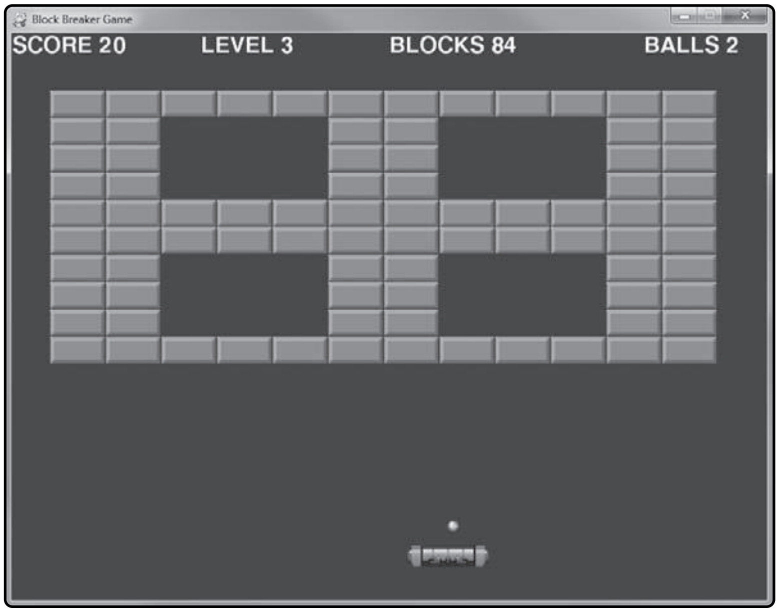


Figure 9.5

Level 3.

```
(3,3,3,3,3,3,3,3,3,3,3,3,
 3,3,0,0,0,3,3,0,0,0,3,3,
 3,3,0,0,0,3,3,0,0,0,3,3,
 3,3,0,0,0,3,3,0,0,0,3,3,
 3,3,3,3,3,3,3,3,3,3,3,3,
 3,3,3,3,3,3,3,3,3,3,3,3,
 3,3,0,0,0,3,3,0,0,0,3,3,
 3,3,0,0,0,3,3,0,0,0,3,3,
 3,3,0,0,0,3,3,0,0,0,3,3,
 3,3,3,3,3,3,3,3,3,3,3,3),
)
```

Loading and Changing Levels

There are three functions in the game for working with the levels. First is `goto_next_level()`, which just increments the level number, makes sure it's within the range of defined levels. Next, the `update_blocks()` function handles the situation when the level has been cleared. The `load_level()` function cycles through the level data to create a sprite group called block_group containing all of the blocks for the current level. Note the use of global variable definitions in these functions—something we have not used very much until now. The global keyword lets the function make changes to a variable which was defined elsewhere in the program.

```
#this function increments the level
def goto_next_level():
    global level, levels
    level += 1
    if level > len(levels)-1: level = 0
    load_level()

#this function updates the blocks in play
def update_blocks():
    global block_group, waiting
    if len(block_group) == 0: #all blocks gone?
        goto_next_level()
        waiting = True
    block_group.update(ticks, 50)

#this function sets up the blocks for the level
def load_level():
    global level, block_image, block_group, levels
    block_image = pygame.image.load("blocks.png").convert_alpha()
    block_group.empty() #reset block group
    for bx in range(0, 12):
        for by in range(0,10):
            block = MySprite()
            block.set_image(block_image, 58, 28, 4)
            x = 40 + bx * (block.frame_width+1)
            y = 60 + by * (block.frame_height+1)
            block.position = x,y
            #read blocks from level data
            num = levels[level][by*12+bx]
            block.first_frame = num-1
            block.last_frame = num-1
            if num > 0: #0 is blank
                block_group.add(block)
```

Initializing the Game

We have a new function in this game to manage initializing Pygame and loading game assets such as bitmap files. The code was beginning to grow by leaps and bounds and will be easier to understand and modify this way.

```
#this function initializes the game
def game_init():
    global screen, font, timer
    global paddle_group, block_group, ball_group
    global paddle, block_image, block, ball

    pygame.init()
    screen = pygame.display.set_mode((800,600))
    pygame.display.set_caption("Block Breaker Game")
    font = pygame.font.Font(None, 36)
    pygame.mouse.set_visible(False)
    timer = pygame.time.Clock()

    #create sprite groups
    paddle_group = pygame.sprite.Group()
    block_group = pygame.sprite.Group()
    ball_group = pygame.sprite.Group()

    #create the paddle sprite
    paddle = MySprite()
    paddle.load("paddle.png")
    paddle.position = 400, 540
    paddle_group.add(paddle)

    #create ball sprite
    ball = MySprite()
    ball.load("ball.png")
    ball.position = 400,300
    ball_group.add(ball)
```

Moving the Paddle

The code to move the paddle taps into the keyboard and mouse events to see when the user is moving the mouse. The mouse can be used to move the paddle left or right, but some may find the game hard to play that way, so the left and right arrow keys are also supported. There is a flag called `waiting` that causes the ball to wait for the player to launch it. This happens when the game first starts or when the player misses the ball (and loses it). Either a mouse button or the Space key will launch the ball when the ball is in the waiting state.

```
#this function moves the paddle
def move_paddle():
    global movex,movey,keys,waiting
    paddle_group.update(ticks, 50)
    if keys[K_SPACE]:
        if waiting:
            waiting = False
            reset_ball()
    elif keys[K_LEFT]: paddle.velocity.x = -10.0
    elif keys[K_RIGHT]: paddle.velocity.x = 10.0
    else:
        if movex < -2: paddle.velocity.x = movex
        elif movex > 2: paddle.velocity.x = movex
        else: paddle.velocity.x = 0
    paddle.X += paddle.velocity.x
    if paddle.X < 0: paddle.X = 0
    elif paddle.X > 710: paddle.X = 710
```

Moving the Ball

There are two functions to manage the ball in the game. `reset_ball()` is a very simple function with just one line, but it is important to make this code reusable because this is where the ball's velocity is defined. Changing the velocity in just one place—this function—is much better than digging through the code for the several instances where the velocity has to be set. The `move_ball()` function does quite a bit of work to make the ball move correctly. The ball needs to move based on its velocity, and also bounce off the edges of the screen. And if the ball falls down below the paddle, then one "ball" or "life" is lost, which could potentially end the game.

```
#this function resets the ball's velocity
def reset_ball():
    ball.velocity = Point(4.5, -7.0)

#this function moves the ball
def move_ball():
    global waiting, ball, game_over, lives
    #move the ball
    ball_group.update(ticks, 50)
    if waiting:
        ball.X = paddle.X + 40
        ball.Y = paddle.Y - 20
    ball.X += ball.velocity.x
    ball.Y += ball.velocity.y
    if ball.X < 0:
        ball.X = 0
        ball.velocity.x *= -1
    elif ball.X > 780:
        ball.X = 780
        ball.velocity.x *= -1
    if ball.Y < 0:
        ball.Y = 0
        ball.velocity.y *= -1
    elif ball.Y > 580: #missed paddle
        waiting = True
        lives -= 1
        if lives < 1: game_over = True
```

Hitting the Paddle

The `collision_ball_paddle()` function handles the collision between the ball and paddle. The ball does not merely bounce off of the paddle by reversing direction. Depending on where the ball hits the paddle, it will deflect away differently, as is typical for this game genre. Hitting the paddle's left side causes the ball to swing to the left, while hitting the right side of the paddle causes the ball to swing right, regardless of the direction it was moving when it hit. This gives the player more control over the ball than just keeping it away from the bottom of the screen.

```
#this function test for collision between ball and paddle
def collision_ball_paddle():
    if pygame.sprite.collide_rect(ball, paddle):
        ball.velocity.y = -abs(ball.velocity.y)
        bx = ball.X + 8
        by = ball.Y + 8
        px = paddle.X + paddle.frame_width/2
        py = paddle.Y + paddle.frame_height/2
        if bx < px: #left side of paddle?
            ball.velocity.x = -abs(ball.velocity.x)
        else: #right side of paddle?
            ball.velocity.x = abs(ball.velocity.x)
```

Hitting the Blocks

The `collision_ball_blocks()` function handles collision detection between the ball and blocks. More importantly, this function handles collision *response*—that is, what happens after the collision takes place. We're using the `pygame.sprite.spritecollideany()` function with the ball and `block_group` passed as parameters, so the ball will be tested against the whole block group. There's some pretty good intelligence in this function that causes the ball to bounce away from the collided block based on the ball's position at the time of collision. To that end, the center of the ball is compared with the center of the paddle. If the ball hits a block from the left or right, it deflects in X but continues in Y. If the ball hits above or below the block, it deflects in Y but continues in X. The result isn't perfect, but offers pretty good gameplay for such a small investment of code.

```
#this function tests for collision between ball and blocks
def collision_ball_blocks():
    global score, block_group, ball

    hit_block = pygame.sprite.spritecollideany(ball, block_group)
    if hit_block != None:
        score += 10
        block_group.remove(hit_block)
        bx = ball.X + 8
        by = ball.Y + 8

        #hit middle of block from above or below?
        if bx > hit_block.X+5 and bx < hit_block.X + hit_block.frame_width-5:
            if by < hit_block.Y + hit_block.frame_height/2: #above?
                ball.velocity.y = -abs(ball.velocity.y)
            else: #below?
                ball.velocity.y = abs(ball.velocity.y)

        #hit left side of block?
        elif bx < hit_block.X + 5:
            ball.velocity.x = -abs(ball.velocity.x)
        #hit right side of block?
        elif bx > hit_block.X + hit_block.frame_width - 5:
            ball.velocity.x = abs(ball.velocity.x)

        #handle any other situation
        else:
            ball.velocity.y *= -1
```

Main Code

The main source code for The Block Breaker Game includes calls to initialize the game, setting up of global variables to their initial values, and of course, the `while` loop. Due to the use of so many functions that have already been covered, the main code is much easier to read and modify, as I'm sure you would agree (compared to the example in previous chapters).

```
#main program begins
game_init()
game_over = False
waiting = True
score = 0
lives = 3
level = 0
load_level()

#repeating loop
while True:
    timer.tick(30)
    ticks = pygame.time.get_ticks()

    #handle events
    for event in pygame.event.get():
        if event.type == QUIT: sys.exit()
        elif event.type == MOUSEMOTION:
            movex,movey = event.rel
        elif event.type == MOUSEBUTTONUP:
            if waiting:
                waiting = False
                reset_ball()
        elif event.type == KEYUP:
            if event.key == K_RETURN: goto_next_level()

    #handle key presses
    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]: sys.exit()

    #do updates
    if not game_over:
        update_blocks()
        move_paddle()
        move_ball()
        collision_ball_paddle()
        collision_ball_blocks()

    #do drawing
    screen.fill((50,50,100))
    block_group.draw(screen)
    ball_group.draw(screen)
    paddle_group.draw(screen)
    print_text(font, 0, 0, "SCORE " + str(score))
    print_text(font, 200, 0, "LEVEL " + str(level+1))
    print_text(font, 400, 0, "BLOCKS " + str(len(block_group)))
    print_text(font, 670, 0, "BALLS " + str(lives))
    if game_over:
        print_text(font, 300, 380, "G A M E   O V E R")
    pygame.display.update()
```

MySprite Update

Before calling it a day, there are some changes to be made to the `MyLibrary.py` file to accommodate some gameplay needs in this chapter project. The `MySprite` class gets an update to make it a little easier to use, as well as give it an important optimization that will cut down on memory usage. When the `MySprite` class was created, it had just a simple `load()` method for loading a bitmap into the `master_image` image. Now, we need a way to recycle an image that is shared by many sprite objects. In the case of The Block Breaker Game in this chapter, we have about 100 blocks per level. Loading the `blocks.png` file into every single one would be a gross waste of memory, not to mention take a bit of time to start up. So, we'll change

MySprite to support image sharing. A modification is made to load(), and a new method called set_image() is added.

```
def load(self, filename, width=0, height=0, columns=1):
    self.master_image = pygame.image.load(filename).convert_alpha()
    self.set_image(self.master_image, width, height, columns)

def set_image(self, image, width=0, height=0, columns=1):
    self.master_image = image
    if width==0 and height==0:
        self.frame_width = image.get_width()
        self.frame_height = image.get_height()
    else:
        self.frame_width = width
        self.frame_height = height
        rect = self.master_image.get_rect()
        self.last_frame = (rect.width//width)*(rect.height//height)-1
    self.rect = Rect(0,0,self.frame_width,self.frame_height)
    self.columns = columns
```

There's one more change to make in MySprite. This change applies to the update() method, and is a small bug fix. The original update() method just uses timing to change animation frames. The bug occurs when the range is changed (first_frame and last_frame), without also changing the frame variable too.

```
def update(self, current_time, rate=30):
      if self.last_frame > self.first_frame:
          #update animation frame number
          if current_time > self.last_time + rate:
              self.frame += 1
              if self.frame > self.last_frame:
                  self.frame = self.first_frame
              self.last_time = current_time
      else:
          self.frame = self.first_frame
```

Summary

This chapter demonstrated the great versatility of Python when it comes to creating and using data with a list or a tuple, and The Block Breaker Game showed how to put these concepts into practice.

Challenges

1. **There's so much potential in The Block Breaker game, where should we begin? Of course, with your own new game level! Add your own new level to the game by adding it to the tuple called `levels` at the top of the source code listing. Hint: Add your new level in *front* of the others rather than after, so you can see it come up first.**
2. **The background of the game is quite boring, just a solid dark blue color. There's so much more that could be done with the background scenery to make the game more interesting and visually appealing. How about an alpha color cycle that causes the background to fade in and out while the game is playing?**
3. **The ball never changes velocity, and this can cause the game to seem to drag at times, not giving the player enough challenge. Add an element of unpredictability to the game by tweaking the ball's velocity just a bit every time it hits the paddle, using a random number.**

CHAPTER 10

Timing and Sound: The Oil Spill Game

Timing is not necessarily a subject closely associated with sound effects or music in a game, but we're going to be using both extensively in this chapter while making a very interesting game. The chapter project is called The Oil Spill Game. While studying timing and new gameplay concepts, you will learn how to load and play sound files.

You will learn how to:

- Load and play a sound using the Pygame mixer
- Use a back buffer to better control drawing
- Create a fast-paced arcade-style game called The Oil Spill Game

Examining The Oil Spill Game

The Oil Spill Game is shown in Figure 10.1. In this game, the player has to clean up an oil spill using a water cannon that fires a high-pressure stream of water at the contaminated areas to clean them up. At least, that's the theory! In reality, we just use the mouse cursor to click on an oil splotch to clean it up! This game uses color alpha channel manipulation to erase the oil splotches and is a good exercise in user input as well as program logic.

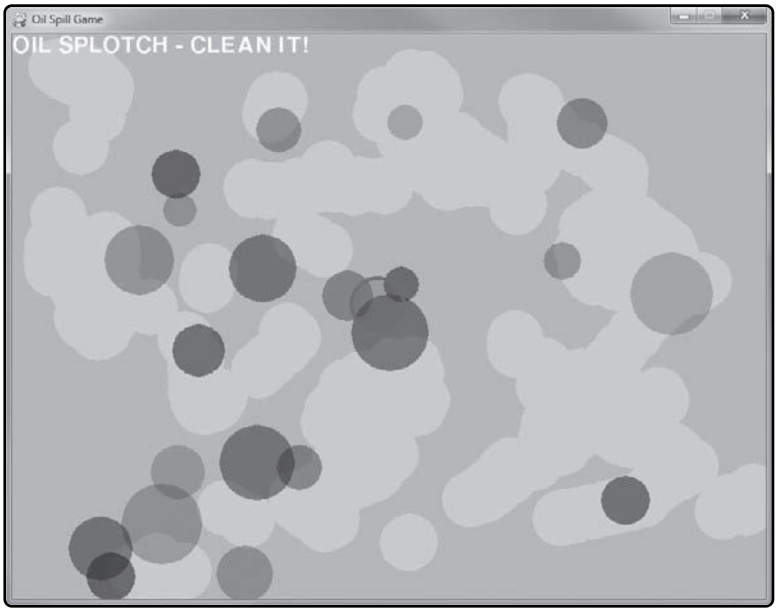


Figure 10.1

The Oil Spill Game uses timing and sound with fun gameplay.

Sound

The audio system we're going to use is included with Pygame in the `pygame.mixer` module. We will learn how to create an audio clip by loading an audio file and then playing the clip in game. Pygame offers some advanced features to control the channels for audio playback, control the mixing of sounds, and even the ability to generate sounds. Pygame does not always initialize the audio mixer, so it's best to initialize it on our own just to be sure our program doesn't crash upon trying to load and play a file. This only needs to be called once:

```
pygame.mixer.init()
```

Loading an Audio File

We use the `pygame.mixer.Sound()` class to load and manage audio objects. Pygame supports two audio file formats: uncompressed WAV and OGG audio files. OGG is a good alternative to MP3 swith no licensing issues. If you wanted to use MP3 in a commercial game, you would have to license the MP3 technology in order to use it. OGG offers similar quality and compression without the licensing! As a result, OGG is a good choice for longer-running music tracks in your game. Of course, you can use OGG for regular short-duration sound effects too. It just tends to be more common to use WAV for shorter audio files and OGG for the longer ones. That is entirely up to you, though. Since WAV files must be uncompressed, you cannot load any WAV file created with a special codec. Since they are uncompressed, the file sizes will tend to be on the large size if they are more than a few seconds in length. Because of the size issue, OGG is recommended for audio clips longer than a few seconds.

```
audio_clip = pygame.mixer.Sound("audio_file.wav")
```

HINT

If you have audio files in some format (like WAV or MP3) that you want to convert to OGG in order to use the format, you will need to convert the files with an audio converter. One good example is free software called Audacity which has advanced audio editing features as well. You can download it from http://audacity.sourceforge.net.

Playing an Audio Clip

The `pygame.mixer.Sound()` constructor returns a `Sound` object. Among the several methods in this class are `play()` and `stop()`. These are easy enough to use, but we're not going to use `Sound` for playback, just for loading and storing the audio data. For playing sounds, we're going to use `pygame.mixer.Channel`. This is a class that offers more versatility than `Sound` for playback. The Pygame audio mixer handles channels internally, so what we do is request an available channel.

```
channel = pygame.mixer.find_channel()
```

This will request an unused channel and return it so we can use it. If there are no available channels, then the mixer returns `None`. Because this may be a problem, if you want to override that default behavior, then pass `True` to `find_channel()` in order to force it to return the lowest-priority channel available.

```
channel = pygame.mixer.find_channel(True)
```

Once we have a channel, we can play a `Sound` object by using the `Channel.play()` method:

```
channel.play(sound_clip)
```

HINT

There are additional features in both the `Sound` and `Channel` classes that you can find in the Pygame documentation online at www.pygame.org/docs/ref/mixer.html.

Building The Oil Spill Game

The Oil Spill Game is an interesting experiment in color manipulation with the use of alpha to cause the oil splotches to disappear as the user "wipes" them with the mouse cursor. This game could be called "stain remover" just as well, and the colors could be changed to any theme for variations on the theme of "oil cleanup."

Gameplay

The oil splotches are indeed sprites, each created as a root `pygame.sprite.Sprite`, and stored in a sprite group with `pygame.sprite.Group`. A custom class inherits from `MySprite` to add some new features required for the game, but they are otherwise `pygame.sprite.Sprite`-based `MySprite` objects. But there is no source artwork—the circles are drawn onto the sprite image at load time.

Timing

Timing is important in The Oil Spill Game. Once every second, a new oil sprite is added to a group that is drawn at a random location on the screen. Each oil sprite has a random radius so that after a while the screen does indeed begin to look like oil is dripping all over it. The trick is to add a new oil sprite only once per second, using timing. There are a few ways to do this, but perhaps the easiest way is with a millisecond timer. First, we create a root time:

```
last_time = 0
```

Then, this root time value is updated whenever the one-second interval is reached using `pygame.time.get_ticks()`:

```
ticks = pygame.time.get_ticks()
if ticks > last_time + 1000:
    add_oil()
    last_time = ticks
```

The key is saving the current ticks value in `last_time` when a timing even occurs. If the ticks value is not saved in `last_time`, then it will only happen once and then never again.

Oil Mess

When the mouse cursor moves around on the screen, circle-based collision detection is used via `pygame.sprite.collide_circle_ratio()` to determine when the mouse cursor is over an oil splotch sprite. Figure 10.2 shows the gameplay when the cursor is over such a sprite.

The oil sprites have a custom image that is created in memory (not loaded from a bitmap file). That image has a dark circle drawn on it with a random radius to represent a single oil splotch.

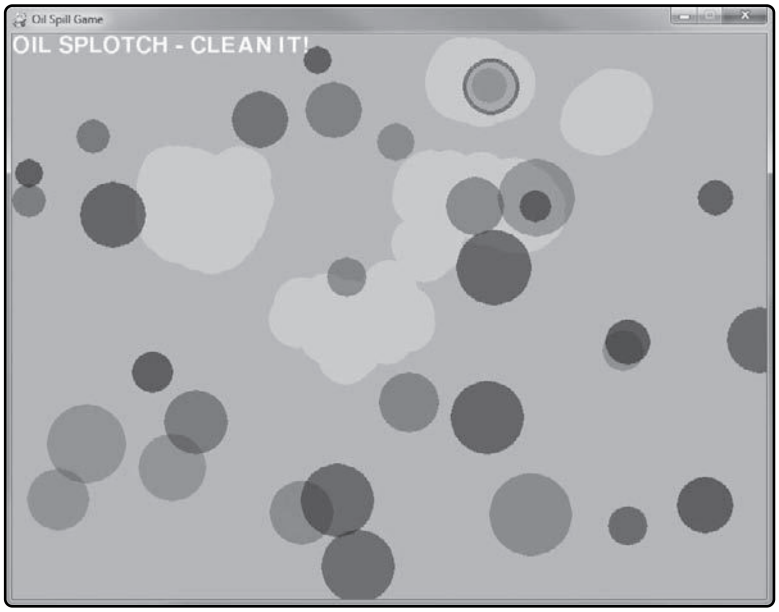


FIGURE 10.2

Identifying oil splotches on the screen with the mouse cursor.

```
image = pygame.Surface((oil.radius,oil.radius)).convert_alpha()
image.fill((255,255,255,0))
oil.fadelevel = random.randint(50,150)
oil_color = 10,10,20,oil.fadelevel
r2 = oil.radius//2
pygame.draw.circle(image, oil_color, (r2,r2), r2, 0)
oil.set_image(image)
```

Cleaning the Oil

By clicking the mouse on an oil splotch, that has the effect of "cleaning" it because the dark circle fades out as it is being cleaned and disappears. This is done by changing the alpha channel color component of the sprite until it is completely invisible. When alpha reaches zero, then the sprite is removed from the group.

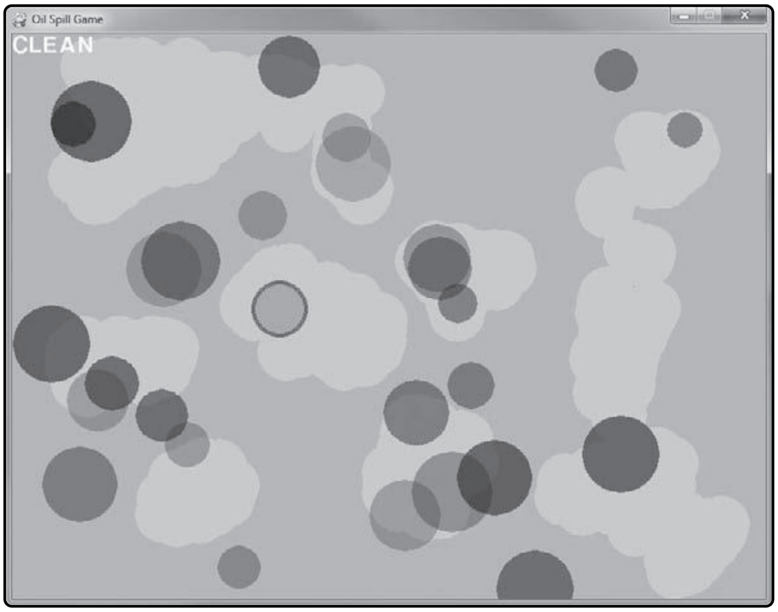


Figure 10.3
Cleaning the oil splotches by "washing" them with the mouse.

Identifying an oil sprite in the group could be done with a group collision method in `pygame.sprite`, but I wanted to get a little more control over how collision response works, so this game iterates the group of oil sprites manually and calls `pygame.sprite.collide_circle_ratio(0.5)`. This makes the collision radius half of the normal value to increase the game's challenge a little bit.

```
    for oil in oil_group:
        if pygame.sprite.collide_circle_ratio(0.5)(cursor, oil):
            oil_hit = oil
```

Washing the Background

Although it is not necessary, to make the game a little more fun, clicking the mouse cursor on the screen causes it to appear to be cleaning the background as well. This helps the player get a feel for the gameplay before any oil splotches even show up. Two colors are defined: `darktan` for the normal background, and `tan` for the cleaned spots:

```
darktan = 190,190,110,255
tan = 210,210,130,255
```

Note that a fourth color component (alpha) is added to these colors. This is required when working with 32-bit color. If you don't specify an alpha channel when creating the color, then it is not available later.

When the user clicks the mouse button anywhere on the screen, a tan circle is drawn to help the player get a feel for what they need to do in the game. It also makes the game more fun.

```
    b1,b2,b3 = pygame.mouse.get_pressed()
    mx,my = pygame.mouse.get_pos()
    pos = (mx+30,my+30)
    if b1 > 0:
        pygame.draw.circle(backbuffer, tan, pos, 30, 0)
```

Source Code

The source code for The Oil Spill Game is not long at all with 147 lines (including blanks and comments), but the source code is rather packed and efficient, so we'll list the entire source code here for easy reference. It is an interesting game. I think there's potential here by designing some gameplay challenges, adding scoring, etc.

```
# Oil Spill Game
# Chapter 10
import sys, time, random, math, pygame
from pygame.locals import *
from MyLibrary import *
darktan = 190,190,110,255
tan = 210,210,130,255

class OilSprite(MySprite):
    def __init__(self):
        MySprite.__init__(self)
        self.radius = random.randint(0,60) + 30 #radius 30 to 90
        play_sound(new_oil)

    def update(self, timing, rate=30):
        MySprite.update(self, timing, rate)

    def fade(self):
        r2 = self.radius//2
        color = self.image.get_at((r2,r2))
        if color.a > 5:
            color.a -= 5
            pygame.draw.circle(self.image, color, (r2,r2), r2, 0)
        else:
            oil_group.remove(self)
            play_sound(clean_oil)

#this function initializes the game
def game_init():
    global screen, backbuffer, font, timer, oil_group, cursor, cursor_group

    pygame.init()
    screen = pygame.display.set_mode((800,600))
    pygame.display.set_caption("Oil Spill Game")
    font = pygame.font.Font(None, 36)
    pygame.mouse.set_visible(False)
    timer = pygame.time.Clock()

    #create a drawing surface
    backbuffer = pygame.Surface((800,600))
    backbuffer.fill(darktan)

    #create oil list
    oil_group = pygame.sprite.Group()

    #create cursor sprite
    cursor = MySprite()
    cursor.radius = 60
    image = pygame.Surface((60,60)).convert_alpha()
    image.fill((255,255,255,0))
    pygame.draw.circle(image, (80,80,220,70), (30,30), 30, 0)
    pygame.draw.circle(image, (80,80,250,255), (30,30), 30, 4)
    cursor.set_image(image)
    cursor_group = pygame.sprite.GroupSingle()
    cursor_group.add(cursor)

#this function initializes the audio system
def audio_init():
    global new_oil, clean_oil

    #initialize the audio mixer
    pygame.mixer.init() #not always called by pygame.init()
    #load sound files
    new_oil = pygame.mixer.Sound("new_oil.wav")
    clean_oil = pygame.mixer.Sound("clean_oil.wav")


def play_sound(sound):
    channel = pygame.mixer.find_channel(True)
    channel.set_volume(0.5)
    channel.play(sound)

def add_oil():
    global oil_group, new_oil

    oil = OilSprite()
    image = pygame.Surface((oil.radius,oil.radius)).convert_alpha()
    image.fill((255,255,255,0))
    oil.fadelevel = random.randint(50,150)
    oil_color = 10,10,20,oil.fadelevel
    r2 = oil.radius//2
    pygame.draw.circle(image, oil_color, (r2,r2), r2, 0)
    oil.set_image(image)
    oil.X = random.randint(0,760)
    oil.Y = random.randint(0,560)
    oil_group.add(oil)

#main program begins
game_init()
audio_init()
game_over = False
last_time = 0

#repeating loop
while True:
    timer.tick(30)
    ticks = pygame.time.get_ticks()

    for event in pygame.event.get():
        if event.type == QUIT: sys.exit()
```

```
keys = pygame.key.get_pressed()
if keys[K_ESCAPE]: sys.exit()

#get mouse input
b1,b2,b3 = pygame.mouse.get_pressed()
mx,my = pygame.mouse.get_pos()
pos = (mx+30,my+30)
if b1 > 0: pygame.draw.circle(backbuffer, tan, pos, 30, 0)

#collision test
oil_hit = None
for oil in oil_group:
    if pygame.sprite.collide_circle_ratio(0.5)(cursor, oil):
        oil_hit = oil
        if b1 > 0: oil_hit.fade()
        break

#add new oil sprite once per second
if ticks > last_time + 1000:
    add_oil()
    last_time = ticks

#draw backbuffer
screen.blit(backbuffer, (0,0))

#draw oil
oil_group.update(ticks)
oil_group.draw(screen)

#draw cursor
cursor.position = (mx,my)
cursor_group.update(ticks)
cursor_group.draw(screen)

if oil_hit: print_text(font, 0, 0, "OIL SPLOTCH - CLEAN IT!")
else: print_text(font, 0, 0, "CLEAN")
pygame.display.update()
```

Summary

This chapter covered the audio system in Pygame via `pygame.mixer` and showed how to load and play a sound file in the context of gameplay events. An interesting game called The Oil Spill Game was used as a backdrop for an audio demonstration program, but it ended up having more interesting color and sprite manipulation code with audio something of an afterthought. Overall, though, there were some valuable concepts and useful source code presented.

Challenges

1. **The Oil Spill Game is more of a demo since there is really no way to win or lose. What do you think would be a good way to improve the replay value without using gimmicks like high score? See if you can come up with ways to improve the gameplay. Perhaps gradually speeding up the rate at which oil splotches are added or some other mechanism?**
2. **Currently, the oil splotches do not move once they have been added to the sprite group. An interesting challenge would be to cause the oil splotches to "smear" as they slowly slide across the screen. How about causing the sprites to move and leave behind a trail of new oil sprites as they go? Be mindful of gameplay balance if you choose to take on this challenge, because it will exponentially add to the number of oil splotches that the player must clean up!**
3. **The game has no way to win or lose. At minimum, count the number of oil sprites and end the game when there are too many (a fairly large number like 180 will be added in 3 minutes of gameplay—remember, it's one sprite per second), and end the game if a limit is reached.**

CHAPTER 11

Program Logic: The Snake Game

This chapter delves into the subject of program logic. How do complex games cause many objects on the screen to move and behave differently without writing code to control each one? Sometimes, it seems that some games use magical source code, because there doesn't seem to be any code to account for what's going on in the game! How is that even possible? If you have ever asked questions such as these while learning game programming, then hopefully this chapter will start to answer some of them. We can only *start* because this is a complex subject. Like most problems in computer science, there are many solutions, not just *one*.

Here is what you will learn in this chapter:

- How to manage a complex array as a list
- How to use timing to slow down a game
- How to add logic to make a game play by itself

Examining The Snake Game

The Snake Game is a classic computer science project dating back *decades*. It is used as a way to help students learn program logic, and so it is a steadfast game concept that remains useful to this day in many a classroom. The premise of the game is this: you are a baby snake (or the head of a snake, if you wish to look at it that way), and must eat food to grow. Every time one piece of food it consumed, the snake grows by one length. When food is eaten, another food is added to a random location. The snake will *keep on* growing until it is so long that the snake literally can't move on the screen any more, and that is when the game ends. Running into any part of the snake's body at any time ends the game. Figure 11.1 shows the game in action.

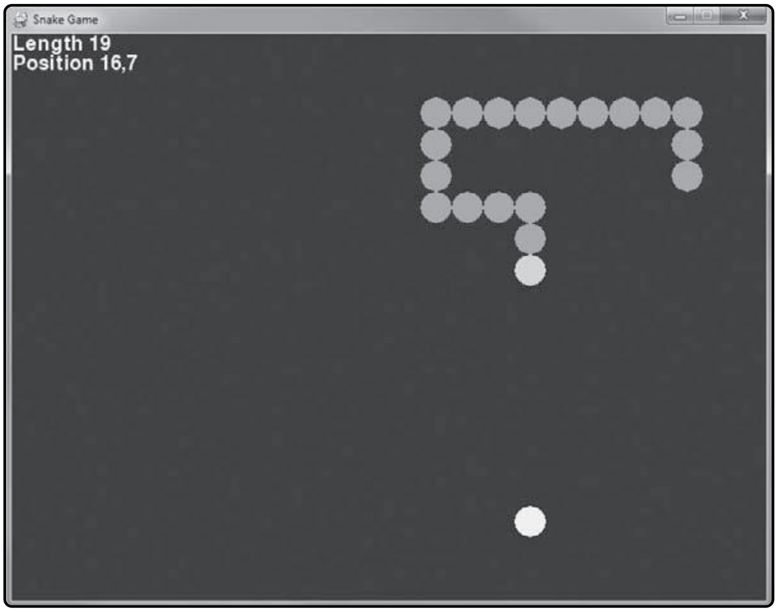


FIGURE 11.1

The Snake Game.

BUILDING THE SNAKE GAME

We're going to build The Snake Game while exploring program logic in this chapter, rather than separating the subjects into theory and application. The snake will be made up of segments, so for an object-oriented programmer, that should get your gears working on a possible class specification. But we also need an array or list to contain and manage the segments. It may be helpful to actually build these custom classes first in order to explore program logic. Like usual, we need our `import` statements to begin:

```
# Snake Game
# Chapter 11
import sys, time, random, math, pygame
from pygame.locals import *
from MyLibrary import *
```

Hatching a Snake—the `SnakeSegment` Class

The key component of the snake is a class called `SnakeSegment`, which derives from `MySprite`. Let's take a quick look and then go over its purpose:

```
class SnakeSegment(MySprite):
    def __init__(self,color=(20,200,20)):
        MySprite.__init__(self)
        image = pygame.Surface((32,32)).convert_alpha()
        image.fill((255,255,255,0))
        pygame.draw.circle(image, color, (16,16), 16, 0)
        self.set_image(image)
        MySprite.update(self, 0, 30) #create frame image
```

Raising a Snake—the `Snake` Class

The `SnakeSegment` class represents one body segment of the snake, represented by a small, green-filled circle on the screen. The segments are linked together like a train, with each segment following the one in front of it. All segments ultimately follow the snake's head, the first segment. Which brings us to the `Snake` class. This class organizes the segments, makes it possible for the segments to follow each other in order, starting with the head. This class also draws the entire snake.

```
class Snake():
    def __init__(self):
        self.velocity = Point(-1,0)
        self.old_time = 0
        head = SnakeSegment((50,250,50))
        head.X = 12*32
        head.Y = 9*32
        self.segments = list()
        self.segments.append(head)
        self.add_segment()
        self.add_segment()

    def update(self,ticks):
        if ticks > self.old_time + 400:
            self.old_time = ticks
            #move body segments
            for n in range(len(self.segments)-1, 0, -1):
                self.segments[n].X = self.segments[n-1].X
                self.segments[n].Y = self.segments[n-1].Y
            #move snake head
            self.segments[0].X += self.velocity.x * 32
            self.segments[0].Y += self.velocity.y * 32

    def draw(self,surface):
        for segment in self.segments:
            surface.blit(segment.image, (segment.X, segment.Y))

    def add_segment(self):
        last = len(self.segments)-1
        segment = SnakeSegment()
        start = Point(0,0)
        if self.velocity.x < 0: start.x = 32
        elif self.velocity.x > 0: start.x = -32
        if self.velocity.y < 0: start.y = 32
        elif self.velocity.y > 0: start.y = -32
        segment.X = self.segments[last].X + start.x
        segment.Y = self.segments[last].Y + start.y
        self.segments.append(segment)
```

Every line of code in the `Snake` class is essential, but the most *significant* lines are shown below. These lines cause the segments of the snake's "body" to follow each other. Every time the head moves one step in the direction specified by the player, all of the segments follow. Each segment replaces the one that was in front of it, while the head segment at the front moves forward in one of the four directions. Every segment is a sprite, ultimately, and each one uses its own position for the purpose of drawing. Every time the head takes a step, which occurs at a fixed time rate, then all of the segments move one space forward.

```
            #move body segments
            for n in range(len(self.segments)-1, 0, -1):
                self.segments[n].X = self.segments[n-1].X
                self.segments[n].Y = self.segments[n-1].Y
```

Feeding the Snake—the `Food` Class

There is a sprite group called `food_group` that could be used to handle more than one food item at a time. That might make for some interesting gameplay! But at present, there is just one food sprite at a time in the game, and that is the goal—to maneuver the snake's head toward the food without hitting the snake's body or any edge of the screen. The `Food` class helps by creating a yellow-filled circle representing the food. You could, of course, replace this manually drawn artwork with custom art to make this a more compelling game, visually, but the purpose of this chapter is not graphics but logic.

```
class Food(MySprite):
    def __init__(self):
        MySprite.__init__(self)
        image = pygame.Surface((32,32)).convert_alpha()
        image.fill((255,255,255,0))
        pygame.draw.circle(image, (250,250,50), (16,16), 16, 0)
        self.set_image(image)
        MySprite.update(self, 0, 30) #create frame image
        self.X = random.randint(0,23) * 32
        self.Y = random.randint(0,17) * 32
```

Initializing the Game

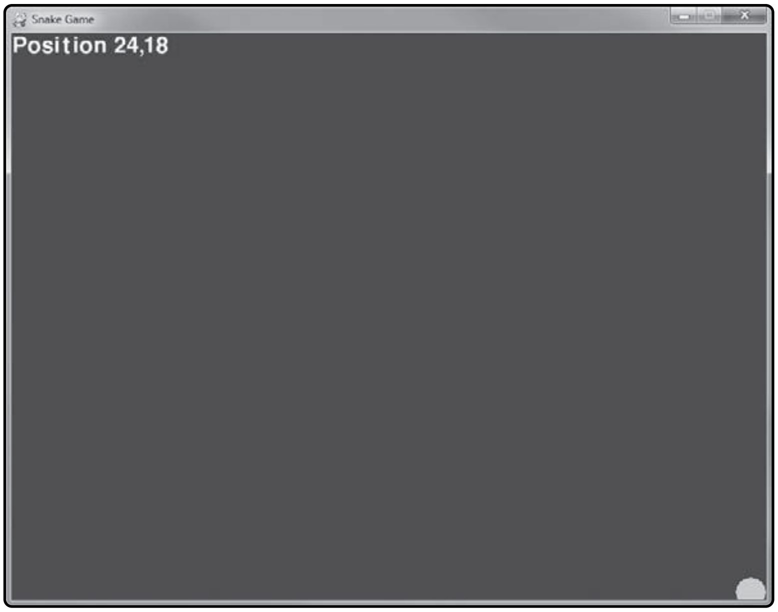
The `game_init()` function handles the initialization code for The Snake Game. Just remember to include a global statement for every global variable you use in any function to avoid bugs or syntax errors. One thing we're using in this game that hasn't been mentioned yet is a back buffer. We actually used a back buffer in the previous chapter example too, but didn't mention it. The back buffer improves the graphics drawing quality of a game by buffering repeated draw calls and then drawing only to the screen once with a whole surface. It might seem wasteful if only a small portion of the screen is being updated, and it is certainly possible to optimize this technique further by looking into Pygame's dirty rectangle rendering capability, but in the interest of simplicity we'll stick with what works since it runs fast enough already.

```
def game_init():
    global screen, backbuffer, font, timer, snake, food_group

    pygame.init()
    screen = pygame.display.set_mode((24*32,18*32))
    pygame.display.set_caption("Snake Game")
    font = pygame.font.Font(None, 30)
    timer = pygame.time.Clock()

    #create a drawing surface
    backbuffer = pygame.Surface((screen.get_rect().width, \
        screen.get_rect().height))

    #create snake
    snake = Snake()
    image = pygame.Surface((60,60)).convert_alpha()
    image.fill((255,255,255,0))
    pygame.draw.circle(image, (80,80,220,70), (30,30), 30, 0)
    pygame.draw.circle(image, (80,80,250,255), (30,30), 30, 4)

    #create food
    food_group = pygame.sprite.Group()
    food = Food()
    food_group.add(food)
```

Did you notice the rather strange resolution values passed to `pygame.display.set_mode()`?

```
    screen = pygame.display.set_mode((24*32,18*32))
```

In order to make the snake move on the screen in a uniform way, given that each snake segment is a 32 × 32 image, we have to divide the screen up into a grid of 32 × 32 spaces or squares. Since the game runs in a window, and not full screen, it doesn't matter what resolution we use, and calculating the width and height to approximate 800 × 600 (it's actually 800 × 576) makes it possible for the snake to move clear around the edge of the game display area. Using our usual 800 × 600 resolution results in the snake not quite fitting on the screen at the very bottom, as shown in Figure 11.2.

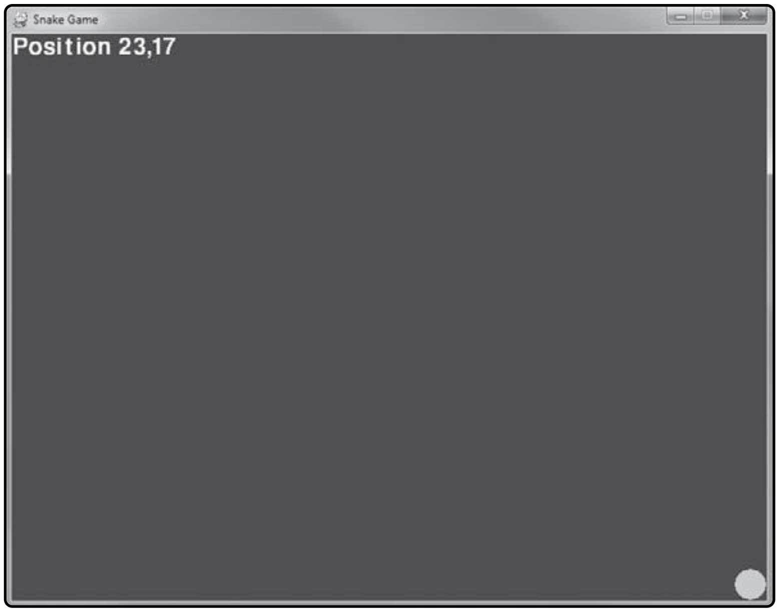


Position 24,18

FIGURE 11.2

The window does not accommodate a 32 × 32 grid in this screenshot.

But by using the calculated resolution of 24 "boxes" across and 18 "boxes" down, we have created a grid of 32 × 32 squares so that the snake can move anywhere on the screen. Taking that into account, we arrive at the resolution shown in Figure 11.3.



```
Position 23,17
```

FIGURE 11.3
The resolution is calculated to make a grid of 32 × 32 squares.

Program Main

The main program code calls `game_init()` and includes the `while` loop that keeps the game running. We'll go over the logic of The Snake Game after the listing.

```
#main program begins
game_init()
game_over = False
last_time = 0

#main loop
while True:
    timer.tick(30)
    ticks = pygame.time.get_ticks()

    #event section
    for event in pygame.event.get():
```

```
    if event.type == QUIT: sys.exit()
keys = pygame.key.get_pressed()
if keys[K_ESCAPE]: sys.exit()
elif keys[K_UP] or keys[K_w]:
    snake.velocity = Point(0,-1)
elif keys[K_DOWN] or keys[K_s]:
    snake.velocity = Point(0,1)
elif keys[K_LEFT] or keys[K_a]:
    snake.velocity = Point(-1,0)
elif keys[K_RIGHT] or keys[K_d]:
    snake.velocity = Point(1,0)

#update section
if not game_over:
    snake.update(ticks)
    food_group.update(ticks)

    #try to pick up food
    hit_list = pygame.sprite.groupcollide(snake.segments, \
        food_group, False, True)
    if len(hit_list) > 0:
        food_group.add(Food())
        snake.add_segment()

    #see if head collides with body
    for n in range(1, len(snake.segments)):
        if pygame.sprite.collide_rect(snake.segments[0], \
            snake.segments[n]):
            game_over = True

    #check screen boundary
    x = snake.segments[0].X//32
    y = snake.segments[0].Y//32
    if x < 0 or x > 24 or y < 0 or y > 17:
        game_over = True

#drawing section
backbuffer.fill((20,50,20))
```

```
    snake.draw(backbuffer)
    food_group.draw(backbuffer)
    screen.blit(backbuffer, (0,0))

    if not game_over:
        print_text(font, 0, 0, "Length " + str(len(snake.segments)))
        print_text(font, 0, 20, "Position " + str(snake.segments[0].X//32)+ \
                   "," + str(snake.segments[0].Y//32))
    else:
        print_text(font, 0, 0, "GAME OVER")

    pygame.display.update()
```

Growth by Eating Food

The game works by moving the snake 32 pixels at a time in the direction specified by the user (via the arrow or W-A-S-D keys). When a new snake segment is added, this direction is taken into account. This method is found in the `Snake` class. Note that the direction of the head is considered when estimating where the new "tail" should be added to the end of the snake.

```
def add_segment(self):
    last = len(self.segments)-1
    segment = SnakeSegment()
    start = Point(0,0)
    if self.velocity.x < 0: start.x = 32
    elif self.velocity.x > 0: start.x = -32
    if self.velocity.y < 0: start.y = 32
    elif self.velocity.y > 0: start.y = -32
    segment.X = self.segments[last].X + start.x
    segment.Y = self.segments[last].Y + start.y
    self.segments.append(segment)
```

This method is called whenever the snake eats food. That happens in the main code under the `while` loop with a call to `pygame.sprite.groupcollide()`. Note that even though `Snake.segments` is defined as a list, it still works like a `pygame.sprite.Group`, because `Group` is derived from a list. So, every segment of the snake is compared to the `food_group` list. Note that the `groupcollide()` function does not remove snake segments, but it *does* remove food (via the fourth parameter). If there is a hit, a new `Food` item is added and a new snake segment is also added. This collision code has the effect of keeping food off the snake's body. Although the head is the only sprite that can "eat" food, if a food sprite is added on a space that is occupied by the snake's body, it is automatically consumed and the snake grows. This could be seen as a bug, but it ends up being a simple way to add a new food item to a valid location on the screen. Figure 11.4 shows the snake about to eat some food.

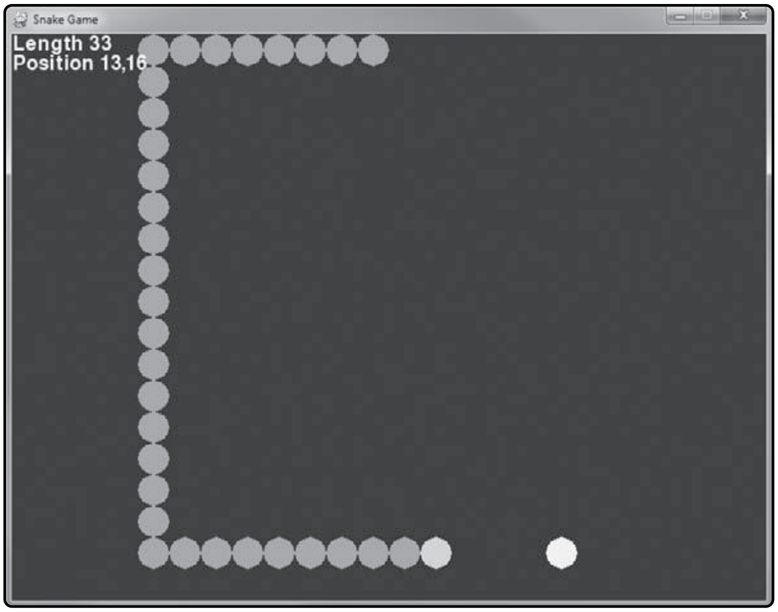


FIGURE 11.4

This already sizable snake is about to eat again.

```
#try to pick up food
hit_list = pygame.sprite.groupcollide(snake.segments, food_group, False, True)
if len(hit_list) > 0:
    food_group.add(Food())
    snake.add_segment()
```

Biting One's Self Is Not Advisable

Two additional sections of code are significant from a logic perspective. First, we check for the case when the head collides with other parts of the snake's body. Note that the first sprite is `snake.segments[0]`, which is the head (the first `SnakeSegment` object added to the list). Next up is `snake.segments[n]` which represents each segment of the body. If the head at any time touches the rest of the body, the game is over. Figure 11.5 shows that state.

```
#see if head collides with body
for n in range(1, len(snake.segments)):
    if pygame.sprite.collide_rect(snake.segments[0], snake.segments[n]):
        game_over = True
```

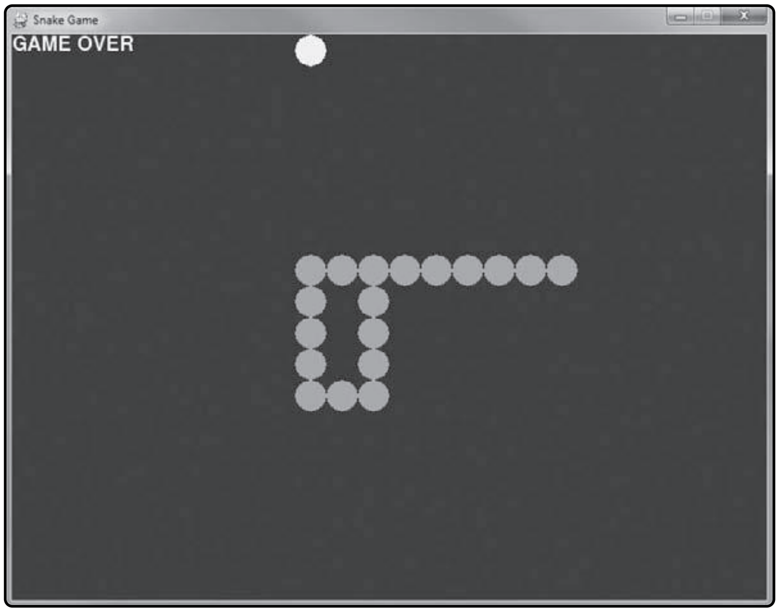


Figure 11.5
The snake head should never touch its body.

Falling off the World

The next major part of the program logic tests for the condition where the snake goes off the edge of the screen. This is another way to lose the game. Note that the position of the snake's head is maintained with pixel precision, but logic uses the "grid" of 32 × 32 squares for this logic.

```
#check screen boundary
head_x = snake.segments[0].X//32
head_y = snake.segments[0].Y//32
if head_x < 0 or head_x > 24 or head_y < 0 or head_y > 17:
    game_over = True
```

Teaching the Snake to Move Itself

Now that we have a fully playable game available, we can get into the real heavy stuff in this chapter—program logic. This code isn't foolproof. The snake will still run into its body segments with this automatic code. But, it's a good exercise in rudimentary game logic and it does a pretty good job of moving toward the food. Figure 11.6 shows the game running in "Auto" mode.

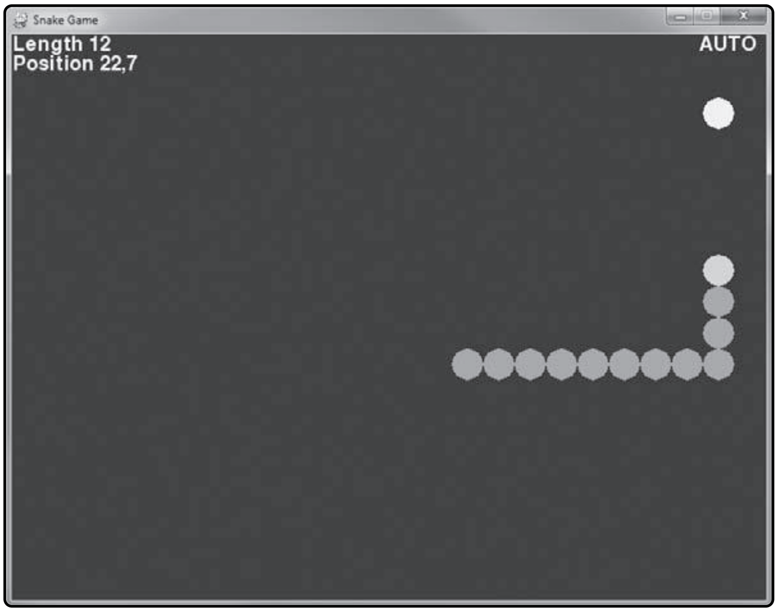


FIGURE 11.6

Teaching the snake to search for food on its own.

Moving Automatically

There are two goals to the automatic snake movement mode of the game (triggered with the Space key). The first goal is to move toward the food. The second goal is for the snake to try not to turn backward and run into itself. We'll define a function called `auto_move()` that will implement these two basic goals with the help of additional helper functions.

```
def auto_move():
    direction = get_current_direction()
    food_dir = get_food_direction()
    if food_dir == "left":
        if direction != "right":
            direction = "left"
    elif food_dir == "right":
        if direction != "left":
            direction = "right"
    elif food_dir == "up":
        if direction != "down":
            direction = "up"
    elif food_dir == "down":
        if direction != "up":
            direction = "down"

    #set velocity based on direction
    if direction == "up": snake.velocity = Point(0,-1)
    elif direction == "down": snake.velocity = Point(0,1)
    elif direction == "left": snake.velocity = Point(-1,0)
    elif direction == "right": snake.velocity = Point(1,0)
```

After adding the modifications to The Snake Game, use the Space key to toggle auto mode.

Getting the Current Direction

The first helper function is called `get_current_direction()`. It works by looking at the first snake body segment to see where it is in relation to the head. Based on this first segment, that tells us the direction the snake is moving. Again, the logic isn't *perfect*, but to make a perfect snake would involve some incredibly complex code that is very hard to write. The code to move the snake around its own body at a certain point starts to look like uber-complex path-finding code found in real-time strategy games. We certainly don't have time for that here, even if it would be really neat to learn how to do that. But I think that's a bit overkill for just a snake game. So, based on that first segment, we find out where the head is moving and return that direction.

```
def get_current_direction():
    global head_x,head_y
    first_segment_x = snake.segments[1].X//32
    first_segment_y = snake.segments[1].Y//32
    if head_x-1 == first_segment_x:   return "right"
    elif head_x+1 == first_segment_x: return "left"
    elif head_y-1 == first_segment_y: return "down"
    elif head_y+1 == first_segment_y: return "up"
```

Moving Toward the Food

The second automation helper function is called `get_food_direction()`. Like the name says, it returns a direction that the snake should move in to get to the food, without regard for running into anything. It just knows which way to go to get to the food. First, the horizontal or X coordinate is checked. Once the snake is lined up horizontally with the food, then it tells the snake to move up or down to get to the food.

```
def get_food_direction():
    global head_x,head_y
    food = Point(0,0)
    for obj in food_group:
        food = Point(obj.X//32,obj.Y//32)
    if head_x < food.x:       return "right"
    elif head_x > food.x:     return "left"
    elif head_x == food.x:
        if head_y < food.y:   return "down"
        elif head_y > food.y: return "up"
```

Other Code Changes

We have some additional changes to make to The Snake Game to give it an automatic playing mode. Let's go over the changes here. There are only a few! Locate the following `update()` method in the `Snake` class and make the changes noted. This makes it so the snake moves really fast when it's running in "auto mode." Although not absolutely necessary, speeding up the snake when in auto mode makes the game much more interesting.

```
    def update(self,ticks):
        global step_time #additional code
        if ticks > self.old_time + step_time: #modified code
```

Down in the main program code after the classes and functions, just before the `while` loop, add the following code to enable "auto play" mode.

```
auto_play = False #additional code added
step_time = 400
```

Inside the `while` loop, near the top where the keyboard handling code is located, add the following code to the key handler:

```
    elif keys[K_SPACE]: #additional code added
        if auto_play:
            auto_play = False
            step_time = 400
        else:
            auto_play = True
            step_time = 100
```

The last change is added to the end of the `while` loop *inside* the `if not game_over:` code block:

```
#additional code added
if auto_play: auto_move()
```

Summary

This chapter showed how to write basic program logic in order to solve problems. The Snake Game was a good experiment for learning program logic, as we were able to make the snake automatically move toward the food. The logic isn't perfect, and the snake will run into itself pretty easily, but it *tries* and does a pretty good job.

Challenges

1. **The Snake Game has so much potential, but that is why it is a favorite among computer science teachers around the world—because it is a tough challenge in working with arrays and using program logic to solve a problem. Let's make it a little easier on the player. Instead of just one food at a time, add more food sprites.**
2. **This challenge is a cinch: Modify the `SnakeSegment` class so that each segment is a different shade of a certain color. Give the snake more coloration but stay within the same "color theme" so it doesn't look random. Can you make it look like a rattlesnake or a king snake or a cobra?**
3. **This challenge is a really tough one, so pay attention: Modify the game so that the game window is four times larger than it is presently. To make this happen, the grid size will be changed from 32 × 32 to 16 × 16 for each square. This will double both the width and height of the grid. The snake body segments will have to be modified so they are also one-fourth their current size. This will make the playing field *huge*!**

This page intentionally left blank

CHAPTER 12

Trigonometry: The Tank Battle Game

In this chapter, we will study the practical use of trigonometry to cause sprites to rotate, move in any direction, and "look" at a target point on the screen. This topic was first introduced back in Chapter 6, "Bitmap Graphics: The Orbiting Spaceship Demo," where we used trigonometry to cause a space ship to rotate in a simulated orbit around a planet. The concept is a powerful one in game programming, so we will explore it further here and learn new ways to put it to use.

You will learn how to:

- Use trigonometry to calculate the velocity at any angle
- Cause a tank turret to point toward a targeting cursor
- Make a really dumb computer A.I. tank that is easy to kill

Examining The Tank Battle Game

The Tank Battle Game, shown in Figure 12.1, involves a tank that can be rotated and moved forward or backward in any direction (by applying angular velocity, one of our most powerful game programming tools). Furthermore, the tank's gun turret rotates independently of the tank chassis, controlled by the mouse that is represented by a crosshair cursor. The player must maneuver the tank with arrow or W-A-S-D keys and use the mouse to fire at enemy tanks. The figure shows the interesting gameplay to be found in The Tank Battle Game. Simply use the mouse cursor to position the crosshair in the direction you wish to fire, and the turret will not only follow the crosshair but will fire in that very same direction!

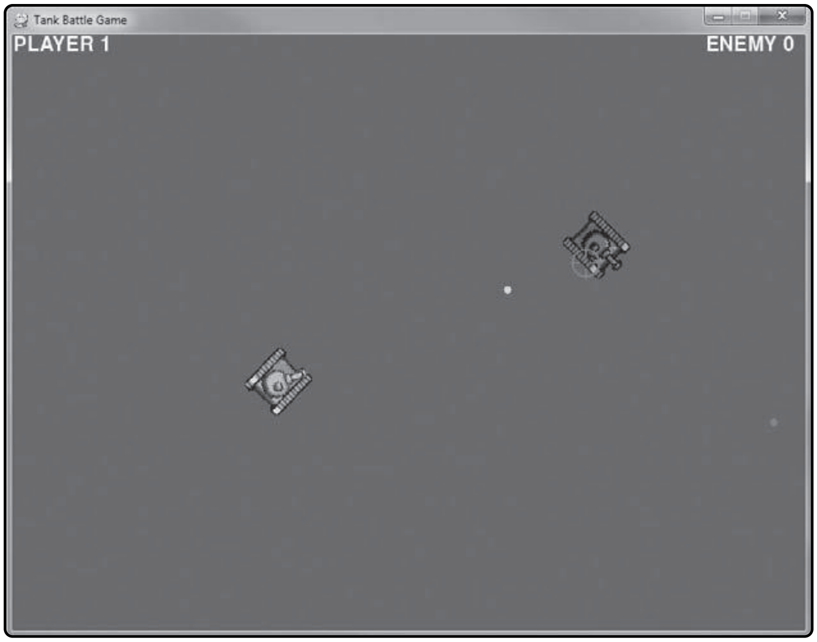


FIGURE 12.1

The Tank Battle Game.

Angular Velocity

Angular velocity describes the velocity (or speed) that an object moves at, represented in terms of X and Y, along any direction among the 360 degrees around the object. The velocity is calculated from the current angle or direction in which the object is facing, with angle 0 degrees being north (up), 90 degrees east (right), 180 degrees south (down), and 270 degrees west (left). However, the velocity need not be limited to the four cardinal directions, because we can calculate angular velocity at any angle, from 0 to 359.999 degrees—and yes, decimals are relevant as well! A partial degree such as 10.5, represents an angle in between 10 and 11 degrees. It is true that trigonometry functions produce values where angle 0 points to the right (east) rather than up (north), so we adjust for that by subtracting 90 degrees from an angle before rotating a sprite.

HINT Pygame uses degrees for sprite rotation. When using trigonometry functions like `math.atan2()` for targeting, be sure to convert the resulting radian angle to degrees with the `math.degrees()` function before using it to rotate a sprite.

Calculating Angular Velocity

You have already seen the trigonometry calculations for angular velocity, but we had not put a name to it at the time (back in Chapter 6). Here are the calculations:

```
Velocity X = cosine( radian angle )
Velocity Y = sine( radian angle )
```

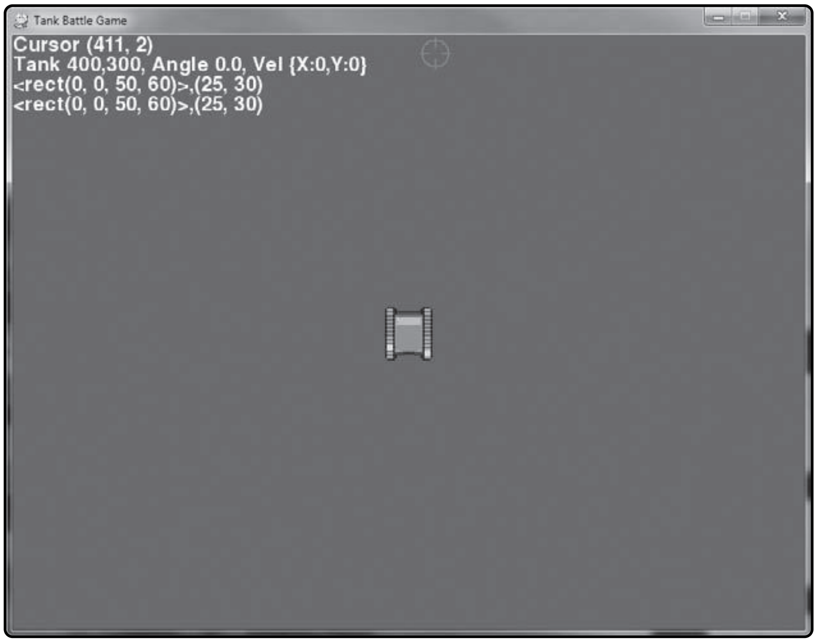
We can codify this in Python like so. Note the use of our custom `Point` class—that is the return type.

```
# calculates velocity of an angle
def angular_velocity(angle):
    vel = Point(0,0)
    vel.x = math.cos( math.radians(angle) )
    vel.y = math.sin( math.radians(angle) )
    return vel
```

Pygame's Goofy Rotation

We first learned how to rotate a sprite in Chapter 5, "Math and Graphics: The Analog Clock Demo," which featured a program that displayed an analog clock with rotating clock pointers. It's pretty easy to rotate a sprite that never moves, but just sits in one place. But when you need to move a sprite and also rotate it, there is a problem. Pygame does not correctly adjust for the change in image size when a sprite is rotated. Take a look at Figure 12.2. This early version of The Tank Battle Game shows a tank chassis (without the turret) used as an example sprite. Note the values printed on the top-left corner. The third line shows the bounding rectangle of the base sprite image (the tank chassis). According to these numbers, the frame has a size of 50 × 60, and the center is at (25,30).

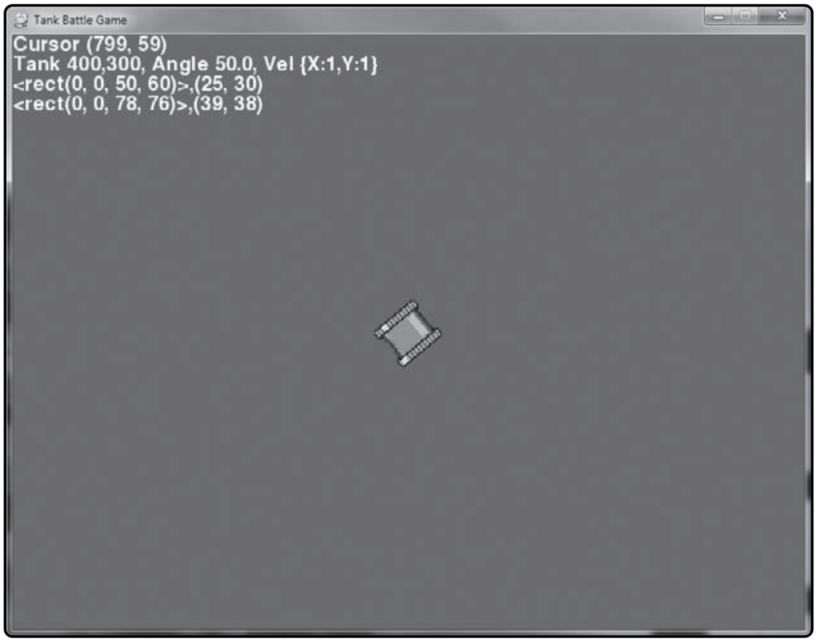
Now compare these numbers with those shown in Figure 12.3. When the sprite is rotated 50 degrees, the bounding rectangle changes to 78 × 76, with a center of (39,38). This is normal for rotation. When a square image is rotated, the corners will require more space at the diagonals than they did before. As a result, the image enlarged by 28 × 16 pixels, and the center moved accordingly. Unfortunately for us, Pygame does not properly take this problem into account like a proper sprite rotation algorithm should (like you find in most other sprite libraries, such as sprite handling in Allegro, DirectX, XNA, and others). But, no matter, we can adjust the sprite ourselves—it just takes a little more work, which is unfortunate given the impression that Python is a fairly quick and easy language to use.



```
Cursor (411, 2)
Tank 400,300, Angle 0.0, Vel {X:0,Y:0}
<rect(0, 0, 50, 60)>,(25, 30)
<rect(0, 0, 50, 60)>,(25, 30)
```

FIGURE 12.2

A sprite's bounding rectangle with no rotation.



```
Cursor (799, 59)
Tank 400,300, Angle 50.0, Vel {X:1,Y:1}
<rect(0, 0, 50, 60)>,(25, 30)
<rect(0, 0, 78, 76)>,(39, 38)
```

FIGURE 12.3

A sprite's bounding rectangle with rotation.

The solution to the bounding rectangle problem with a rotated sprite is to shift the image by the amount of change in dimensions from the normal image to the rotated image. We *could* change the image property internally in the `MySprite` class so that `self.image` represents a rotated sprite at the adjusted position, and then allow `pygame.sprite.Sprite` and `pygame.sprite.Group` to continue to draw the image for us. But, that ended up being even more work than just taking control of the update and draw process on our own. So, that is the direction I'll take here. Let's see how to do it.

First, we'll need a scratch image for rotation. I've just called it `scratch`, but you could call it `rotated_image` to be more descriptive. Rather than modify `MySprite`, I've opted instead to create a new class that derives from `MySprite`, and it is called `Tank`.

Moving Forward and Backward at Any Angle

Angular velocity can be used for more than just moving a bullet or arrow in any direction. We can also use it to move a game object (like a tank) forward and backward based on user input. This is where gameplay becomes really interesting! We can actually rotate our game sprite left or right, and then move it forward or backward based on the direction it's pointing. This is brilliant for vehicles like cars and tanks, because it makes them move more realistically! We can even make a sprite *slow down* at whatever direction it's moving. I have used this technique to good effect in space combat games where the ship can rotate in any direction and fire while still going in the direction of its momentum. If you wish to make a sci-fi game, like an *Asteroids* clone, you will be able to with the information gleaned in this chapter. For good measure, here again is the `target_angle()` function added to MyLibrary in the previous chapter:

```
# calculates angle between two points
def target_angle(x1,y1,x2,y2):
    delta_x = x2 - x1
    delta_y = y2 - y1
    angle_radians = math.atan2(delta_y,delta_x)
    angle_degrees = math.degrees(angle_radians)
    return angle_degrees
```

We can use this function, along with a tank's rotation angle, to cause one of our tanks in The Tank Battle Game to move forward or backward at any angle. Now, this could be used to move the enemy tanks as well, but to keep the already rather complex code down to a manageable level, the enemy tanks will just move in one direction and fire only forward (without rotating the turret). If you want the enemy tanks to also rotate their turrets and move around more realistically, that would be a good upgrade to the game that you may wish to make (see the Challenges at the end of the chapter).

Put into use, this function produces an angle we can use immediately in our gameplay code. First, we get the velocity, then we update the position of the sprite using that velocity. The only problem is, `pygame.sprite.Sprite` (from which our `MySprite` class is derived) uses integer properties for the sprite's position (a `Rect`, actually). This obliterates our velocity code! Unfortunately, we have to write a workaround for this small problem. The solution is a new property added to our own new custom class (which will be called `Tank`), called `float_pos`. We just have to be sure to update `MySprite.position` with the values in `float_pos` before drawing.

```
self.velocity = angular_velocity(angle)
self.float_pos.x += self.velocity.x
self.float_pos.y += self.velocity.y
```

Despite this capability, The Tank Battle Game just keeps the tanks moving forward at a constant velocity to simplify the gameplay. During testing, I found that targeting with the mouse cursor and rotating and moving the tank all at once was asking a bit too much from the player. Instead, the tank moves forward and you may turn left or right.

When in doubt, open up the MyLibrary.py file to see how `Sprite` handles updates and position properties.

Improved Angle Wrapping

While we're adding new code to MyLibrary.py, here's a minor tweak to the `wrap_angle()` function that keeps it in bounds when the angle is negative:

```
# wraps a degree angle at boundary
def wrap_angle(angle):
    return abs(angle % 360)
```

Building The Tank Battle Game

I think that is enough information to go on at this point in order to get started on The Tank Battle Game. This is the most complex game we have developed so far in the book. The complexity is not necessarily due to the game being *complex*, as in, *gameplay*, but rather in the significant amount of workaround code we have to write to get our sprites to behave correctly when Pygame does not quite handle things in a logical way. It happens. But we just need to learn how these workarounds work and be aware of any issues that crop up as a result.

The Tanks

By far the largest class in this game is the Tank class. The reason for its size is that Tank is fully self-contained. All of the code for initialization and logic is here in the class, including gameplay code, rather than outside in the main program. The Tank.update() function is the largest one we have seen to date. The class was primarily designed for the player's tank, not enemy tanks. But it was easily adapted to enemy tanks with a wrapper class called EnemyTank. Tank, of course, inherits from MySprite. An interesting feature of Tank is that the chassis and turret rotate independently of each other, which makes for some very interesting gameplay. The arrow keys or W-A-S-D keys are used to move the tank, while the turret tracks the mouse cursor using target_angle(). While moving the tank around, just point with the mouse cursor at your target and the turret automatically points at it! The gameplay is actually quite fun for as limited as it is.

HINT

All of the artwork for the game is included in the resource files for this chapter.

The Tank Constructor

Now let's start with the Tank constructor. As is to be expected, a call to the MySprite constructor comes first. Since we are wrapping MySprite, it makes sense to also handle image loading for the sprite rather than leaving that entirely in the hands of the programmer (although the default tank sprite filenames may be replaced).

```
class Tank(MySprite):
    def __init__(self,tank_file="tank.png",turret_file="turret.png"):
        MySprite.__init__(self)
        self.load(tank_file, 50, 60, 4)
        self.speed = 0.0
        self.scratch = None
        self.float_pos = Point(0,0)
        self.velocity = Point(0,0)
        self.turret = MySprite()
        self.turret.load(turret_file, 32, 64, 4)
        self.fire_timer = 0
```

The Tank Update Function

This is one monster of an update() function, but it does have some gameplay code as well as class code. The code in Tank.update() handles the difficult problem of movement so our main program remains cleaner and easier to understand. The first section of code here in update() creates the scratch image for the tank's rotation. Remember, MySprite already does animation, and our tank sprite is animated with four frames (see Figure 12.4). I removed the turret from Ari Feldman's original tank sprite, separating the two so that the turret could be moved independently from the chassis.

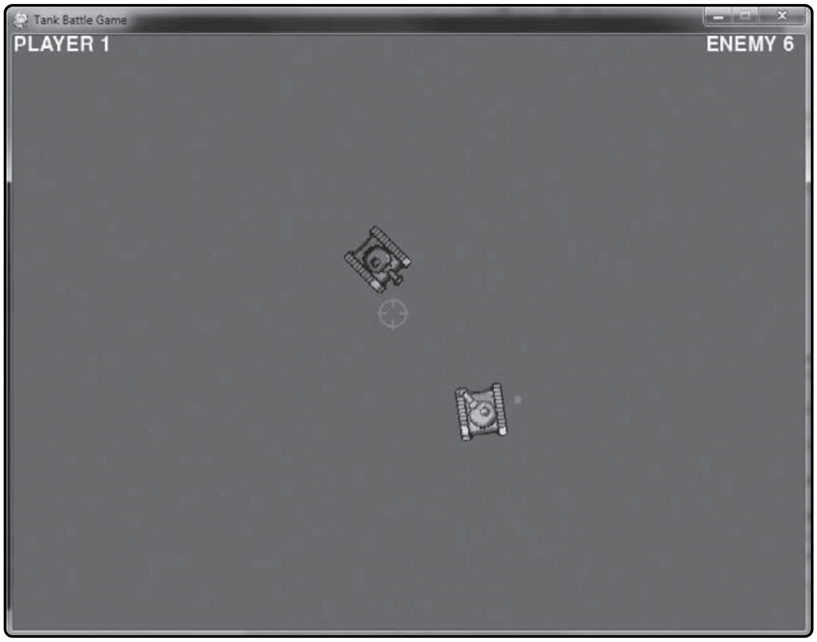


Figure 12.4

The four frames of animation for the tank sprite.

The Real World

Credit for the tank artwork featured in this chapter goes to Ari Feldman. Check out his site for more free game sprites at http://www.widgetworx.com/widgetworx/portfolio/spritelib.html. The old website, www.flyingyogi.com, still forwards to this new location at the time of this writing.

The turret sprite image is shown in Figure 12.5. Although there are four frames for the turret, The Tank Battle Game only uses the first frame. If you want to use the other frames, that would be an interesting upgrade to the game since the turret looks quite good when animated as a shot is fired.

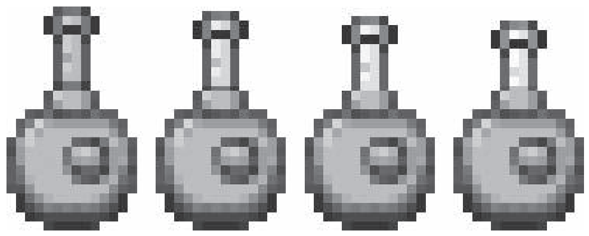


FIGURE 12.5
The tank turret also has four frames of animation, but are not used in this example.

```
def update(self,ticks):
    #update chassis
    MySprite.update(self,ticks,100)
    self.rotation = wrap_angle(self.rotation)
    self.scratch = pygame.transform.rotate(self.image, -self.rotation)
    angle = wrap_angle(self.rotation-90)
    self.velocity = angular_velocity(angle)
    self.float_pos.x += self.velocity.x
    self.float_pos.y += self.velocity.y

    #warp tank around screen edges (keep it simple)
    if self.float_pos.x < -50: self.float_pos.x = 800
    elif self.float_pos.x > 800: self.float_pos.x = -50
    if self.float_pos.y < -60: self.float_pos.y = 600
    elif self.float_pos.y > 600: self.float_pos.y = -60

    #transfer float position to integer position for drawing
    self.X = int(self.float_pos.x)
    self.Y = int(self.float_pos.y)

    #update turret
    self.turret.position = (self.X,self.Y)
    self.turret.last_frame = 0
    self.turret.update(ticks,100)
    self.turret.rotation = wrap_angle(self.turret.rotation)
    angle = self.turret.rotation+90
    self.turret.scratch = pygame.transform.rotate(self.turret.image,
        -angle)
```

The Tank Draw Function

The `Tank.draw()` function has a lot of work to do, because it must take into account animation frames and rotation of the chassis as well as the problematic turret. The turret sprite really is difficult to manage. It has to move along with the chassis (the main tank sprite), and also rotate to aim toward the mouse cursor. Because targeting produces results that somewhat conflicts with Pygame's normal behavior, we have to adjust the position of the sprite as it is drawn without affecting the base position of the sprite (otherwise it would appear to shake or wobble on the screen).

```
    def draw(self,surface):
        #draw the chassis
        width,height = self.scratch.get_size()
        center = Point(width/2,height/2)
        surface.blit(self.scratch, (self.X-center.x, self.Y-center.y))
        #draw the turret
        width,height = self.turret.scratch.get_size()
        center = Point(width/2,height/2)
        surface.blit(self.turret.scratch, (self.turret.X-center.x,
            self.turret.Y-center.y))
```

The String Override

We have a minor override of the `__str__()` function so that information about the tank can be easily returned as a string and used to print out the status of the object for debugging purposes. Note that the base `MySprite` string function is called first, and an extra value is just added to the end of it. This way, we still have the basic information coming from `MySprite`, while adding new properties as needed.

```
    def __str__(self):
        return MySprite.__str__(self) + "," + str(self.velocity)
```

The EnemyTank Class

The `EnemyTank` class is derived from `Tank`, adding some specialty code to make it work a little better (simple A.I. code). The enemy tank only moves in one direction and fires once per second in the same direction. This is really simplistic behavior, but we have to start somewhere and most of the work in this game has been devoted to the player's tank controls. So, think of the enemy tank at this point as just a clay pigeon, or a moving target that poses little threat, but which has *potential* to be an intelligent foe.

```
class EnemyTank(Tank):
    def __init__(self,tank_file="enemy_tank.png",turret_file="enemy_turret.png"):
        Tank.__init__(self,tank_file,turret_file)

    def update(self,ticks):
        self.turret.rotation = wrap_angle(self.rotation-90)
        Tank.update(self,ticks)

    def draw(self,surface):
        Tank.draw(self,surface)
```

The Bullets

The Bullet class helps with projectile management in the game. There are also three additional helper functions outside of the class that make firing a bullet from any tank rather easy to do.

```
class Bullet():
    def __init__(self,position):
        self.alive = True
        self.color = (250,20,20)
        self.position = Point(position.x,position.y)
        self.velocity = Point(0,0)
        self.rect = Rect(0,0,4,4)
        self.owner = ""

    def update(self,ticks):
        self.position.x += self.velocity.x * 10.0
        self.position.y += self.velocity.y * 10.0
        if self.position.x < 0 or self.position.x > 800 \
            or self.position.y < 0 or self.position.y > 600:
                self.alive = False
        self.rect = Rect(self.position.x, self.position.y, 4, 4)

    def draw(self,surface):
        pos = (int(self.position.x), int(self.position.y))
        pygame.draw.circle(surface, self.color, pos, 4, 0)

def fire_cannon(tank):
    position = Point(tank.turret.X, tank.turret.Y)
    bullet = Bullet(position)
    angle = tank.turret.rotation
    bullet.velocity = angular_velocity(angle)
    bullets.append(bullet)
    play_sound(shoot_sound)
    return bullet

def player_fire_cannon():
    bullet = fire_cannon(player)
    bullet.owner = "player"
    bullet.color = (30,250,30)

def enemy_fire_cannon():
    bullet = fire_cannon(enemy_tank)
    bullet.owner = "enemy"
    bullet.color = (250,30,30)
```

Main Code

Now that we have all of the prerequisite classes and functions done, we can address the main gameplay code of The Tank Battle Game. At this point, there will be more initialization code than gameplay code, but we'll go over each section.

The Header Code

The header for the program is always helpful to see spelled out, even if the import list has not changed in quite some time.

```
# Tank Battle Game
# Chapter 12
import sys, time, random, math, pygame
from pygame.locals import *
from MyLibrary import *
```

Game Initialization

Our consistent use of `game_init()` to initialize Pygame, the display, and global variables helps to organize the source code of the game and make it much more readable. Just be sure to add any new globals to the global definitions at the top of the function.

```
#this function initializes the game
def game_init():
    global screen, backbuffer, font, timer, player_group, player, \
           enemy_tank, bullets, crosshair, crosshair_group

    pygame.init()
    screen = pygame.display.set_mode((800,600))
    backbuffer = pygame.Surface((800,600))
    pygame.display.set_caption("Tank Battle Game")
    font = pygame.font.Font(None, 30)
    timer = pygame.time.Clock()
    pygame.mouse.set_visible(False)

    #load mouse cursor
    crosshair = MySprite()
    crosshair.load("crosshair.png")
    crosshair_group = pygame.sprite.GroupSingle()
    crosshair_group.add(crosshair)

    #create player tank
    player = Tank()
    player.float_pos = Point(400,300)

    #create enemy tanks
    enemy_tank = EnemyTank()
    enemy_tank.float_pos = Point(random.randint(50,760), 50)
    enemy_tank.rotation = 135

    #create bullets
    bullets = list()
```

The Audio Functions

The Tank Battle Game has rudimentary audio in the form of two sound clips—one for firing a bullet, another for hitting a target. Audio has not played a large role in any of the examples, but suffice it to say, sound is extremely important in a production game, and even a little music wouldn't hurt. Of course, for a small sample project like this even a small sound clip here and there is a welcome improvement to what is otherwise purely a graphics and gameplay demo.

```
# this function initializes the audio system
def audio_init():
    global shoot_sound, boom_sound

    #initialize the audio mixer
    pygame.mixer.init()
    #load sound files
    shoot_sound = pygame.mixer.Sound("shoot.wav")
    boom_sound = pygame.mixer.Sound("boom.wav")
# this function uses any available channel to play a sound clip
def play_sound(sound):
    channel = pygame.mixer.find_channel(True)
    channel.set_volume(0.5)
    channel.play(sound)
```

Gameplay Code

The gameplay code (the main code of the game) follows. This code is rather short considering how much gameplay there is with the tanks, due to the fact that a lot of that code is found in the `Tank` class itself. That's good news for our gameplay code here, because any duplicate tanks would have to be updated manually even if in a list. Pay particular attention to the bullet update code, which is where collision detection between the bullets and tanks occurs. There is an identifier in the `Bullet` class called `Bullet.owner` that is set to either "player" or "enemy" to aid in collision testing. Without this distinction, it is very hard to keep tanks from blowing themselves up as soon as they fire a bullet! Figure 12.6 shows the two tanks facing off. The player's bullets are green, while the enemy's bullets are red.

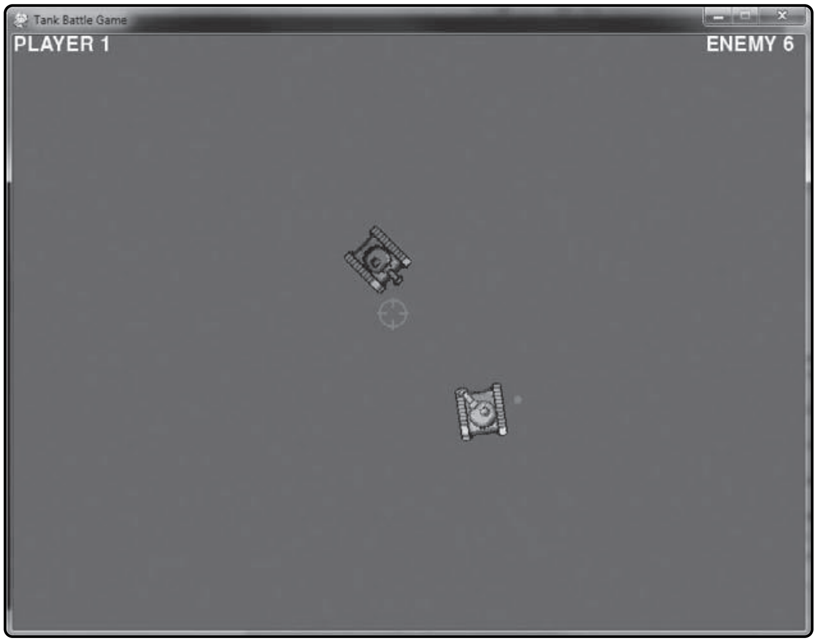


FIGURE 12.6

The player is getting hit by the enemy tank!

```
#main program begins
game_init()
audio_init()
game_over = False
player_score = 0
enemy_score = 0
last_time = 0
mouse_x = mouse_y = 0

#main loop
while True:
    timer.tick(30)
    ticks = pygame.time.get_ticks()

    #reset mouse state variables
    mouse_up = mouse_down = 0
    mouse_up_x = mouse_up_y = 0
    mouse_down_x = mouse_down_y = 0

    #event section
    for event in pygame.event.get():
        if event.type == QUIT: sys.exit()
        elif event.type == MOUSEMOTION:
            mouse_x,mouse_y = event.pos
            move_x,move_y = event.rel
        elif event.type == MOUSEBUTTONDOWN:
            mouse_down = event.button
            mouse_down_x,mouse_down_y = event.pos
        elif event.type == MOUSEBUTTONUP:
            mouse_up = event.button
            mouse_up_x,mouse_up_y = event.pos

    #get key states
    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]: sys.exit()

    elif keys[K_LEFT] or keys[K_a]:
        #calculate new direction velocity
        player.rotation -= 2.0

    elif keys[K_RIGHT] or keys[K_d]:
        #calculate new direction velocity
        player.rotation += 2.0

    #fire cannon!
    if keys[K_SPACE] or mouse_up > 0:
        if ticks > player.fire_timer + 1000:
            player.fire_timer = ticks
            player_fire_cannon()

    #update section
    if not game_over:
        crosshair.position = (mouse_x,mouse_y)
        crosshair_group.update(ticks)

        #point tank turret toward crosshair
        angle = target_angle(player.turret.X,player.turret.Y,
            crosshair.X + crosshair.frame_width/2,
            crosshair.Y + crosshair.frame_height/2)
        player.turret.rotation = angle

        #move tank
        player.update(ticks)

        #update enemies
        enemy_tank.update(ticks)
        if ticks > enemy_tank.fire_timer + 1000:
            enemy_tank.fire_timer = ticks
            enemy_fire_cannon()

        #update bullets
        for bullet in bullets:
            bullet.update(ticks)
            if bullet.owner == "player":
                if pygame.sprite.collide_rect(bullet, enemy_tank):
                    player_score += 1
                    bullet.alive = False
                    play_sound(boom_sound)
            elif bullet.owner == "enemy":
                if pygame.sprite.collide_rect(bullet, player):
                    enemy_score += 1
                    bullet.alive = False
                    play_sound(boom_sound)

    #drawing section
    backbuffer.fill((100,100,20))

    for bullet in bullets:
        bullet.draw(backbuffer)
    enemy_tank.draw(backbuffer)
    player.draw(backbuffer)
    crosshair_group.draw(backbuffer)

    screen.blit(backbuffer, (0,0))

    if not game_over:
        print_text(font, 0, 0, "PLAYER " + str(player_score))
        print_text(font, 700, 0, "ENEMY " + str(enemy_score))
    else:
        print_text(font, 0, 0, "GAME OVER")

    pygame.display.update()

    #remove expired bullets
    for bullet in bullets:
        if bullet.alive == False:
            bullets.remove(bullet)
```

Summary

This chapter showed how to use awesome trigonometry functions to make game sprites behave like the sprites in a professional game. We learned advanced concepts like targeting and demonstrated how it works with a game that allows the user to rotate a tank turret to point at the mouse cursor. This is the basis for even more advanced behaviors such as chasing and evading another object, and finding a path around obstacles.

Challenges

1. Upgrade The Tank Battle Game so the enemy tanks move more like the player's tank, using rotation and angular velocity to chase after the player.
2. Modify the enemy tanks so that they can rotate their turrets and fire just like the player's tank can. This will require some additional A.I. logic code, though, so be prepared for some extra work if you take on this challenge!
3. Add a "Health" property to the `Tank` class and draw a health bar for each tank. Every successful hit reduces the health. Scoring only occurs when health drops to zero and the tank is killed. Then, re-spawn the tank with full health at a new random location.

CHAPTER 13

Random Terrain: The Artillery Gunner Game

This chapter shows how to create a random side-view terrain generator. The terrain could be used for a number of different game designs, from a side-scrolling platformer game, to a shoot-em-up, and others. We'll use it to make an Artillery Gunner game. This game will demonstrate how to really use a height map terrain system effectively.

In this chapter, you will learn:

- About height map terrain.
- How to create an awesome Artillery Gunner game.
- How to decimate the computer player because it has dumb A.I. code

Examining The Artillery Gunner Game

The Artillery Gunner Game is shown in Figure 13.1. Let's get started working on it already!

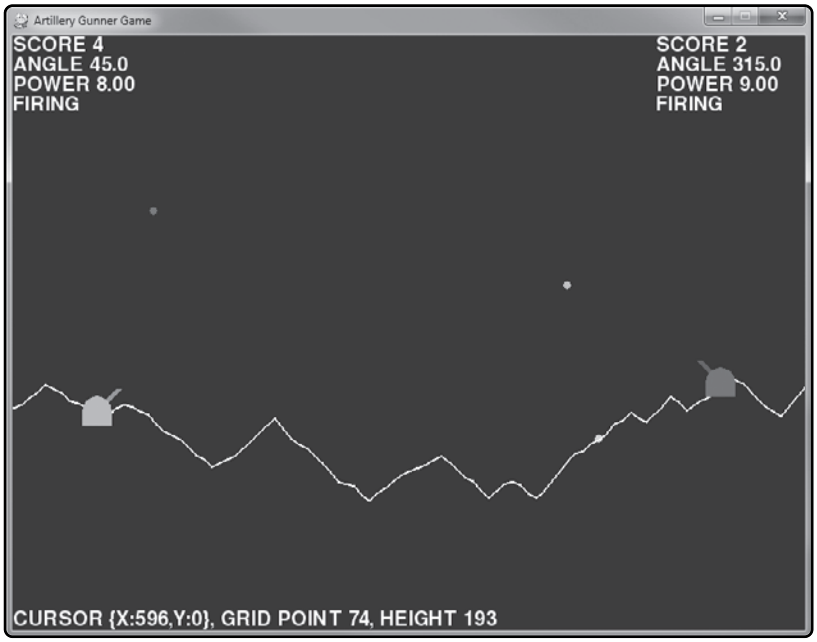


Figure 13.1

The Artillery Gunner Game.

Creating the Terrain

The first thing we'll focus on is the terrain system for the game. The terrain will be 2D, of course, and represented from the side-view perspective. The terrain will be a list of height map points, each representing a height value from the bottom of the screen upward. Despite the screen bottom being at location 600 (using our default window size of 800 × 600), it works because the terrain is drawn from the bottom up rather than from the top down. So, a height value of 100 will be drawn at 600 − 100 or **500** on the screen as a Y coordinate.

Defining the Height Map

Let's create the height map data and draw it on the screen with small circles to get a feel for how it will work. We will pass three parameters to the `Terrain` class constructor, representing the minimum height, the maximum height, and the total points across (which may also be considered the *granularity* or *smoothness* of the terrain). We can calculate the distance between each point by dividing the window width by the number of points. So, for instance, if there will be 100 points, then they will be separated by 800 / 100 or **8** pixels each.

Terrain Class, First Edition

Here is the start of the class that will do just this small amount of work so far. Note that `height_map` is the name of the list containing height values.

```
class Terrain():
    def __init__(self, min_height, max_height, total_points):
        self.min_height = min_height
        self.max_height = max_height
        self.total_points = total_points+1
        self.grid_size = 800 / total_points
        self.height_map = list()
        height = (self.max_height + self.min_height) / 2
        self.height_map.append( height )
        for n in range(total_points):
            height = random.randint(min_height, max_height)
            self.height_map.append( height )

    def draw(self, surface):
        for n in range( 1, self.total_points ):
            #draw circle at current point
            x_pos = int(n * self.grid_size)
            pos = (x_pos, height)
            color = (255,255,255)
            pygame.draw.circle(surface, color, pos, 4, 1)
```

Drawing the Terrain

We'll write a small test program to see what the terrain looks like at this early stage. The result of the code is shown in Figure 13.2. From within `game_init()`, the terrain object is created:

```
    #create terrain
    terrain = Terrain(100, 400, 20)
```

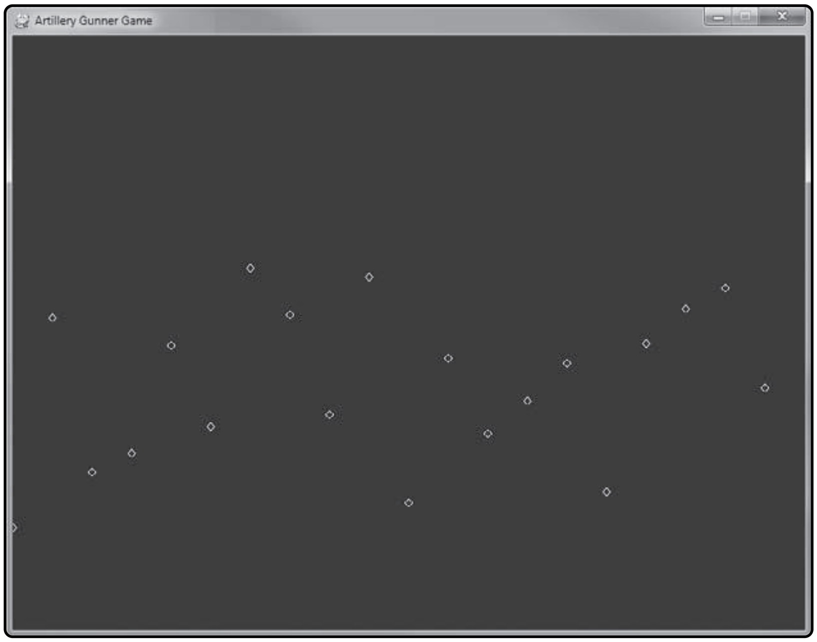


Figure 13.2

First attempt at height map terrain.

In the drawing portion of the code, we call on `terrain.draw()` to have it draw itself to the back buffer. Then the usual screen updates take place.

```
#drawing section
backbuffer.fill((20,20,120))
terrain.draw(backbuffer)
screen.blit(backbuffer, (0,0))
pygame.display.update()
```

Connecting the Dots

Do you find it somewhat confusing, seeing the terrain in this manner? It does look very much like a scatter chart. If we connect the points, it will look more like what we'd expect. Let's do that now. Add the lines in bold.

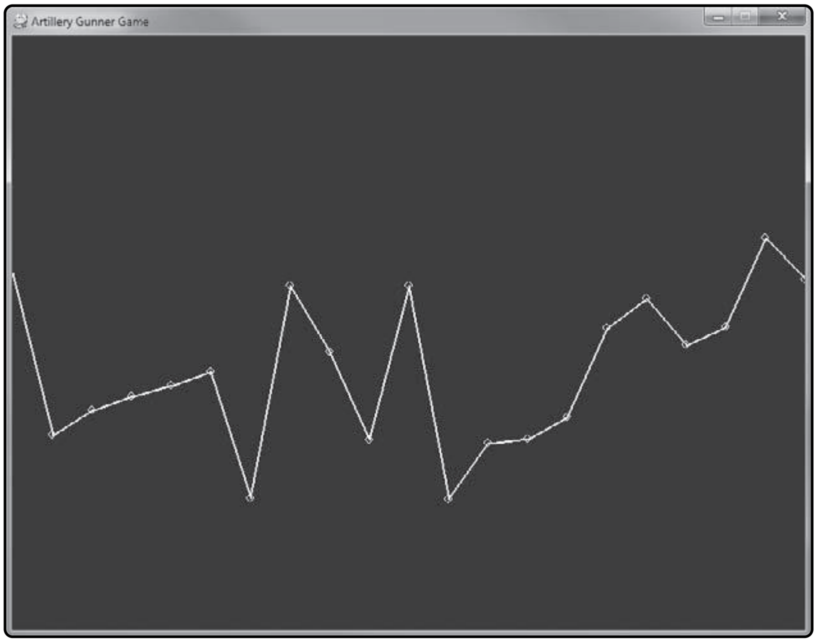


Figure 13.3
Connecting the height map data points.

```
def draw(self, surface):
    last_x = 0
    for n in range( 1, self.total_points ):
        #draw circle at current point
        height = 600 - self.height_map[n]
        x_pos = int(n * self.grid_size)
        pos = (x_pos, height)
        color = (255,255,255)
        pygame.draw.circle(surface, color, pos, 4, 1)
        #draw line from previous point
        last_height = 600 - self.height_map[n-1]
        last_pos = (last_x, last_height)
        pygame.draw.line(surface, color, last_pos, pos, 2)
        last_x = x_pos
```

TRICK

Experiment! Try out different height map random ranges and grid sizes to see what happens! See if you can come up with some interesting new gameplay ideas.

At this point, I recommend you do what I just did, experiment with different height map random ranges and grid sizes to see what happens. The default range is a height of 100 to 400. Try different values to see what results you get. For instance, Figure 13.4 shows a random terrain height map array generated with a height range of only 100 to 120 and a grid size of 50 points.

```
    terrain = Terrain(100, 120, 50)
```

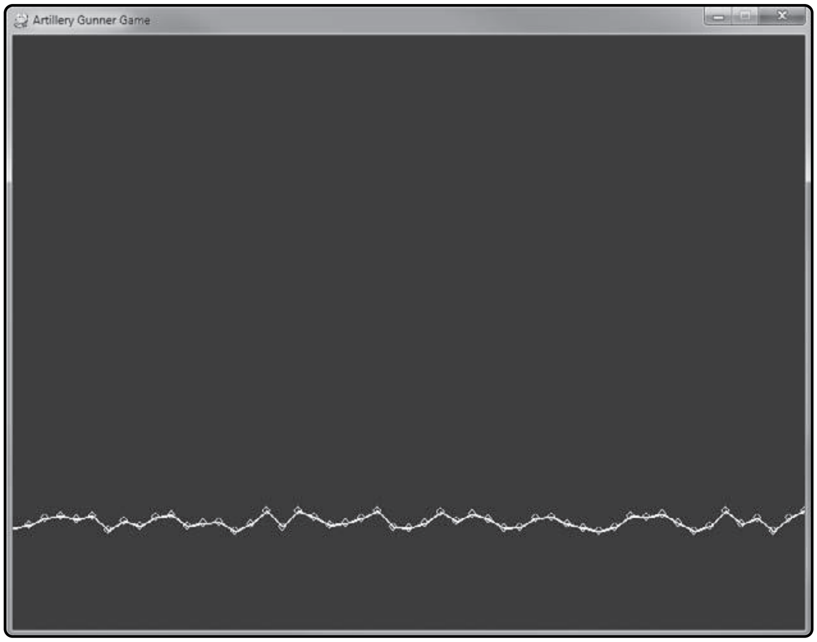


FIGURE 13.4
A height map with a random range of 100–120 and 100 points.

Let's try another. Note that all of these random terrain examples *could* be used in the game! They might not give us very good granularity for doing special effects like creating craters in the ground, but it would work nonetheless.

To demonstrate how radical the terrain can get, try this one out. Yikes! Figure 13.6 shows the result. It is starting to look like a sample in an audio editing program.

```
    terrain = Terrain(100, 500, 100)
```

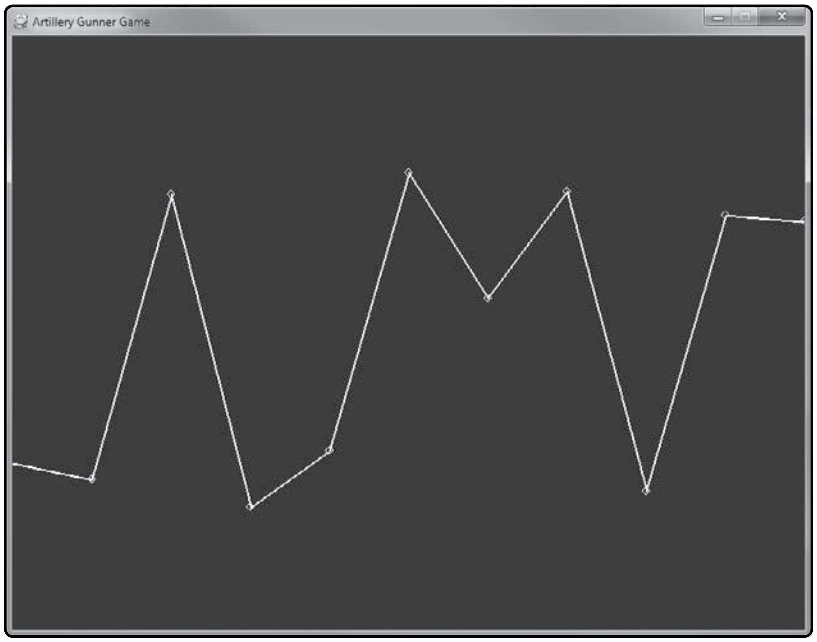


Figure 13.5

A height map with a random range of 100-500 and only 10 points.

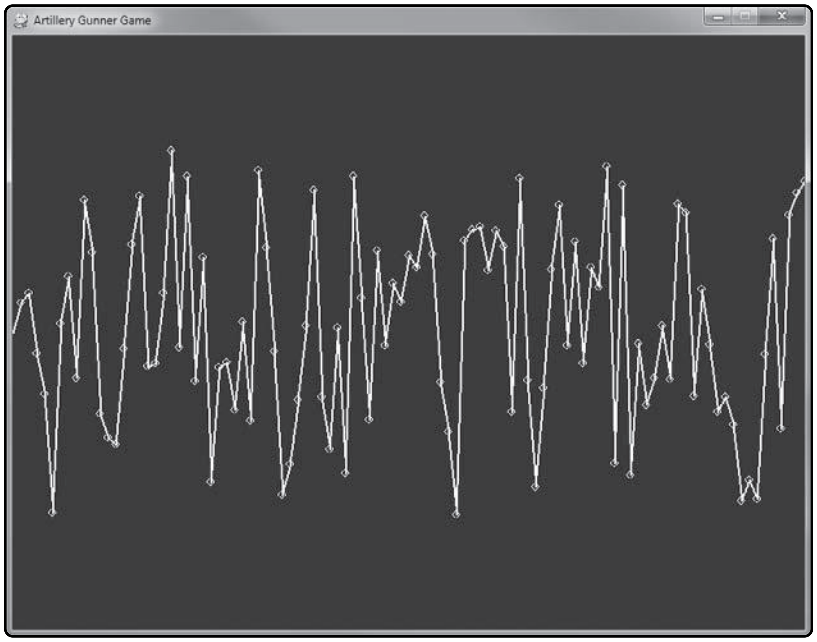


Figure 13.6

A jagged terrain created from a range of 100-500 and 100 points.

Smoothing the Terrain

We're off to a good start, but playing with manual settings is too much work. What we need is an algorithm that will smooth out the terrain, give it a less jagged appearance. The way to do that— or, I should say, *one way* to do that, as there are many ways!—is to make each point a random amount up or down relative to the height of the previous point. This will get rid of the jaggies. Let's try it out. First, it will be helpful to move the terrain generation code out of the constructor and into a reusable method. This way we can experiment by re-generating the terrain several times per run without stopping and restarting the program over and over. This also means we have to clear out the height map each time because of the way the list is constructed.

Here are the changes to the constructor, and the new `generate()` method. The smoothing algorithm works like this. A random run-length value gets a small random value. The terrain will continue to randomly move in the same basic direction (up or down) during that run. When the run is completed, a new run and direction is generated. This causes the points to stay fairly close to each other, getting rid of the wild jagged edges we saw previously. See Figure 13.7.

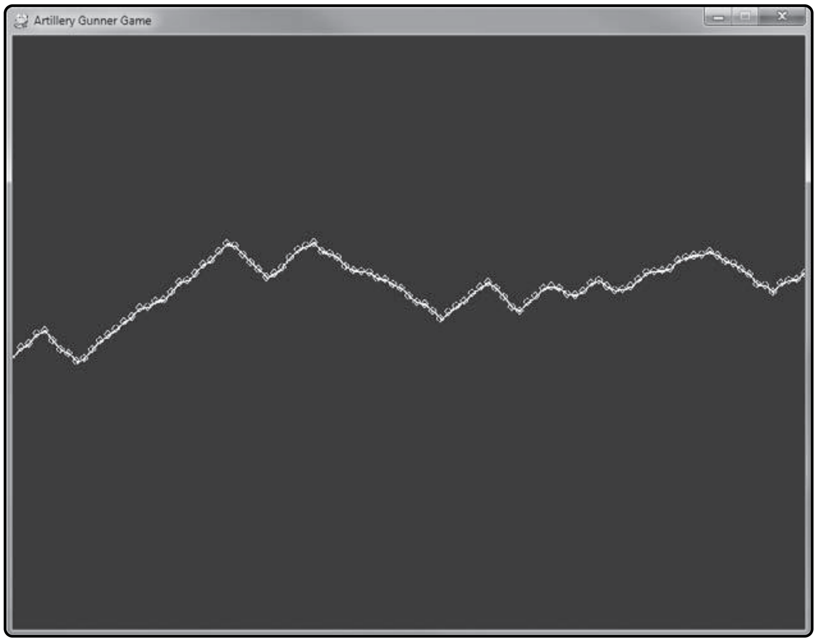


FIGURE 13.7

The height map is much smoother with the new algorithm.

```
def __init__(self, min_height, max_height, total_points):
    self.min_height = min_height
    self.max_height = max_height
    self.total_points = total_points+1
    self.grid_size = 800 / total_points
    self.height_map = list()
    self.generate()

def generate(self):
    #clear list
    if len(self.height_map)>0:
        for n in range(self.total_points):
            self.height_map.pop()

    #first point
    last_x = 0
    last_height = (self.max_height + self.min_height) / 2
    self.height_map.append( last_height )
    direction = 1
    run_length = 0

    #remaining points
    for n in range( 1, self.total_points ):
        rand_dist = random.randint(1, 10) * direction
        height = last_height + rand_dist
        self.height_map.append( int(height) )
        if height < self.min_height: direction = -1
        elif height > self.max_height: direction = 1
        last_height = height
        if run_length <= 0:
            run_length = random.randint(1,3)
            direction = random.randint(1,2)
            if direction == 2: direction = -1
        else:
            run_length -= 1
```

Just to show what this algorithm can do, here is a new screenshot in Figure 13.8. Compare this with the previous one and note the ranges involved. This particular terrain (in both figures) has a range of 50 to 500, and is comprised of 100 points. Experiment to see what different results you get!

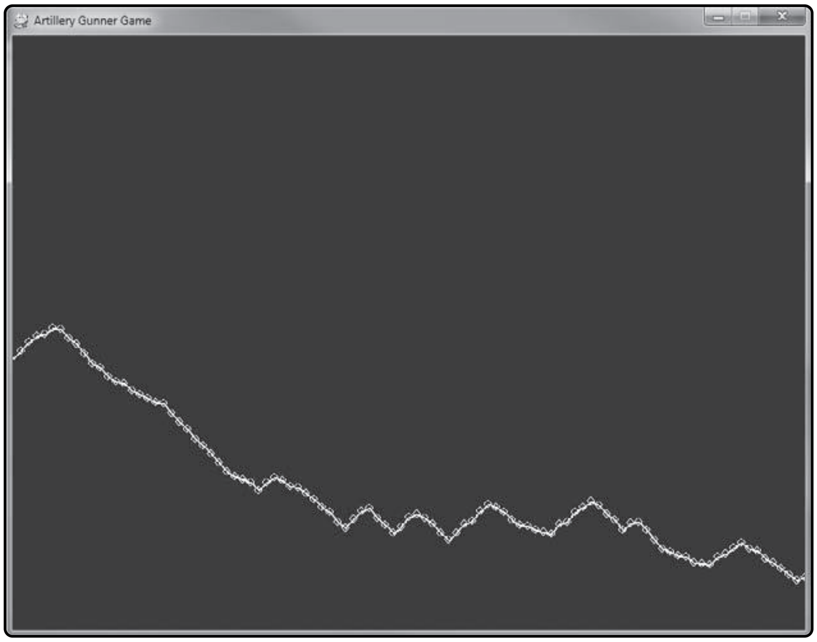


FIGURE 13.8

This shows the wide variation of height maps that can still be generated.

Locating Grid Points

Now that the circles have served their purpose, we can get rid of them and just draw the lines. But first, before doing that, I want to show you an important calculation. We need to be able to tell where the points are located on the height map based on screen coordinates. The key to doing that is by using the `grid_size` property. It is calculated in the `Terrain` constructor based on the `total_points` parameter (the number of divisions for the terrain height map across the screen). More total points results in a smaller grid and more detailed (less jagged) terrain.

To determine the grid point on the terrain corresponding to the cursor position, we first get the cursor position as a pair of integers called `mouse_x` and `mouse_y` via the `MOUSEMOTION` event. The grid point is then calculated based on the mouse cursor's X position, divided by the terrain grid size.

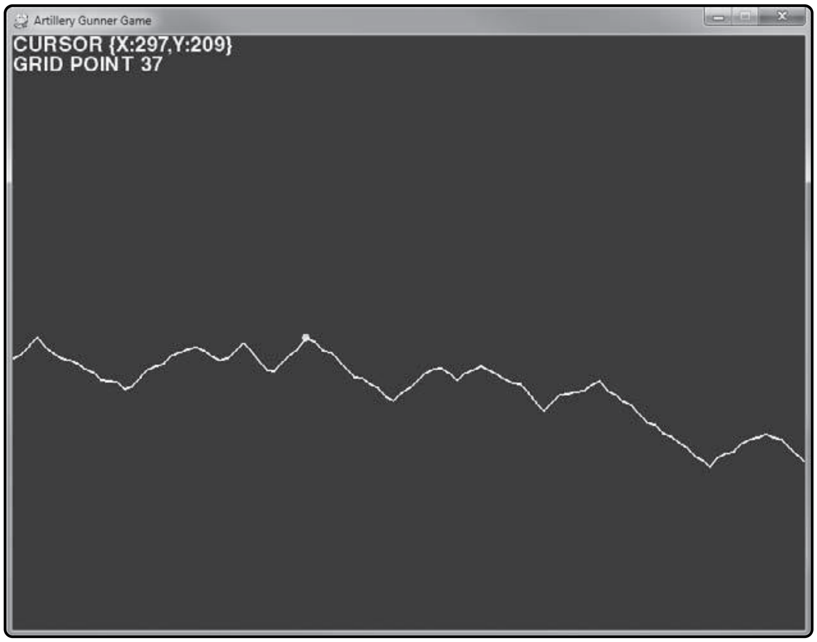
```
    grid_point = int(mouse_x / terrain.grid_size)
```

With the grid_point variable a global, we can use it in a modified version of the Terrain.draw() method to highlight the terrain point corresponding to the mouse cursor position. This point will seem to move over the terrain based on the cursor's X position. Presto, game-ready height map terrain is ready to go! In the modified draw() below, the original pygame.draw.circle() has been commented out, and another has been inserted within a condition. When the current grid point is the point corresponding to the mouse cursor, via the grid_point variable, then we can draw it as a solid circle. The result is shown in Figure 13.9.

```
def draw(self, surface):
    last_x = 0
    for n in range( 1, self.total_points ):
        #draw circle at current point
        height = 600 - self.height_map[n]
        x_pos = int(n * self.grid_size)
        pos = (x_pos, height)
        color = (255,255,255)
        #pygame.draw.circle(surface, color, pos, 4, 1)
        if n == grid_point:
            pygame.draw.circle(surface, (0,255,0), pos, 4, 0)
        #draw line from previous point
        last_height = 600 - self.height_map[n-1]
        last_pos = (last_x, last_height)
        pygame.draw.line(surface, color, last_pos, pos, 2)
        last_x = x_pos
```

This helper method (in Terrain) will make it easier to calculate the height of any point on the terrain:

```
def get_height(self,x):
    x_point = int(x / self.grid_size)
    return self.height_map[x_point]
```



```
CURSOR {X:297,Y:209}
GRID POINT 37
```

Figure 13.9

The green dot shows the terrain grid point at the cursor position.

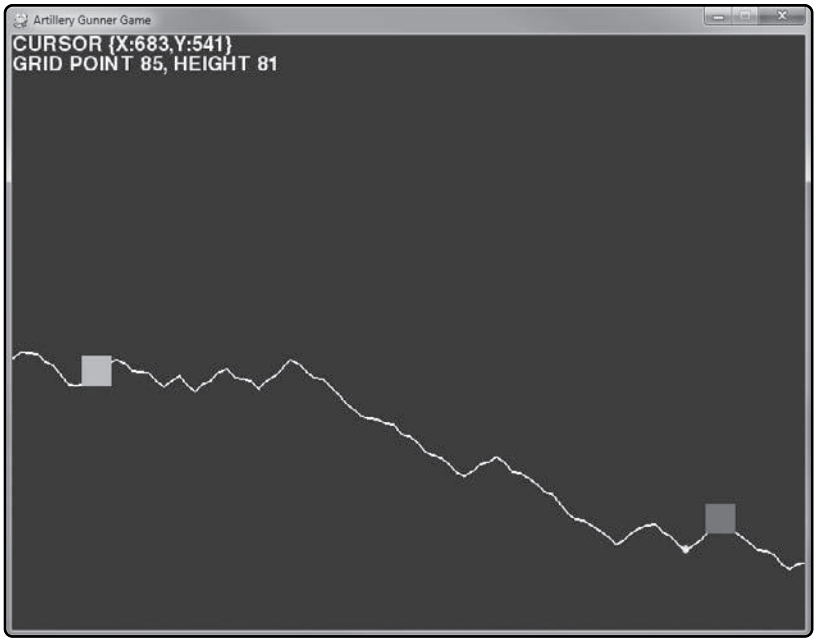
Artillery Cannons

Now that we can get the height of any point on the terrain, we can use that information to position the artillery guns. The way this will work is, there will be two guns—one controlled by the player on the left, and one controlled by computer A.I. on the right. They will be positioned near the left and right edges of the screen.

Placing the Cannons

The artillery guns will be represented with a box and a line coming out of it for the turret. I'm a programmer, not an artist! But seriously, the game works fine with drawing code, but you can replace it with bitmaps if you wish. The cannons will be positioned horizontally at X location 70 on the left (for the player), and X location 700 on the right (for the computer). The cannons don't have turrets yet, just a colored box representing the gun's position. See Figure 13.10.

```
def draw_player_cannon(surface,position):
    color = (30,220,30)
    rect = Rect(position.x, position.y, 30, 30)
    pygame.draw.rect(surface, color, rect, 0)


def draw_computer_cannon(surface,position):
    color = (220,30,30)
    rect = Rect(position.x, position.y, 30, 30)
    pygame.draw.rect(surface, color, rect, 0)
```



```
CURSOR {X:683,Y:541}
GRID POINT 85, HEIGHT 81
```

FIGURE 13.10

Positioning the artillery cannons on the terrain using height map data.

Drawing the Turrets

We need a way to keep track of two factors for the artillery cannon: 1) The power which determines how far a shell will go, and 2) The angle at which the shell will be fired from the cannon. To keep the computer A.I. simple, it will have a turret with a fixed angle, and it will just fire at the player using random power (or range) values. We'll work on that code in a minute. First, let's just draw the turrets and let the player change the angle. We will just use the Up and Down arrow keys for the turret angle. Here is the new `draw_player_cannon()` function that uses `angular_velocity` to calculate the end point of the turret.

```
def draw_player_cannon(surface,position):
    #draw turret
    turret_color = (30,180,30)
    start_x = position.x + 15
    start_y = position.y + 15
    start_pos = (start_x, start_y)
    vel = angular_velocity( wrap_angle(player_cannon_angle-90) )
```

```
end_pos = (start_x + vel.x * 30, start_y + vel.y * 30)
pygame.draw.line(surface, turret_color, start_pos, end_pos, 6)
#draw body
body_color = (30,220,30)
rect = Rect(position.x, position.y+15, 30, 15)
pygame.draw.rect(surface, body_color, rect, 0)
pygame.draw.circle(surface, body_color, (position.x+15,position.y+15), 15, 0)
```

The code for the `draw_computer_cannon()` function is similar, but the turret is pointing in the other direction (left), and it is red rather than green.

Firing the Cannons

We use similar code to fire a cannon to what was used to draw the turret, since we want the shell to fire at the same angle as the turret. There are quite a few new globals in this section of code, which was necessary at this point. `player_cannon_power` is set using the Left and Right arrow keys, with a range of 0.0 to 10.0. `player_cannon_position` of course represents the position of the player's artillery cannon on the screen, with the height value already accounted for. `player_shell_position` will represent the current position of a shell fired from the gun when `player_firing` is `True`; otherwise, this code is just ignored. The player can only fire when a shell is not already in the air. Similar code is used to fire the computer's cannon shells at the player.

```
angle = wrap_angle( player_cannon_angle - 90 )
player_shell_velocity = angular_velocity( angle )
player_shell_velocity.x *= player_cannon_power
player_shell_velocity.y *= player_cannon_power
player_shell_position = player_cannon_position
player_shell_position.x += 15
player_shell_position.y += 15
```

Shots Are A'Flyin

Once the player has fired, the `player_firing` variable is `True`, and that causes the shell (a little green circle) to launch from the player's gun. Likewise, when `computer_firing` is `True`, a little red circle launches from the computer's gun. The position of the shell is updated like so:

```
if player_firing:
    player_shell_position.x += player_shell_velocity.x
    player_shell_position.y += player_shell_velocity.y
```

Now, the real trick to firing a projectile and simulating the arc and causing it to fall on a target is the use of Y velocity. As the shell is flying through the air, the Y velocity starts off a negative value, like −8.0. This represents the number of pixels the little shell will move per frame. Every frame, that Y velocity is *increased* by a small amount. Over time, the Y velocity will eventually be 0.0, and it will be flying horizontally for a brief moment. Then, as Y velocity continues to increase, it will become positive and the shell will begin to arc downward toward the ground.

```
if player_shell_velocity.y < 10.0:
    player_shell_velocity.y += 0.1
```

Of course, we want the shells to stop if they hit the ground (the height map terrain) or go off the screen. Here is how we determine when a shell has hit the ground:

```
height = 600 - terrain.get_height(player_shell_position.x)
if player_shell_position.y > height:
    player_firing = False
```

And here is how we make sure a shell does not continue to fly outside the boundaries of the screen:

```
if player_shell_position.x < 0 or player_shell_position.x > 800:
    player_firing = False
if player_shell_position.y < 0 or player_shell_position.y > 600:
    player_firing = False
```

Computer Firing

We want the computer to at least put up the *appearance* of trying to beat the player to make the game somewhat fun. I use the word *trying* because we are not going to spend the time calculating the exact angle and power to hit the player's artillery cannon, even though that is possible. This is old-school rocket science! Believe it or not, this is what computers were invented to do in the first place. Back in World War II, the first electro-mechanical computers were put to use crunching artillery gun calculations for the Allies.

Now, the computer's artillery cannon has a turret that does not move, unlike the player's turret which *can* move. So, the angle does not ever change. It will remain fixed at 315 degrees (which is the opposite side of 45 degrees). The power factor is another matter, however. This is a value that we will randomly generate for each shot. If the computer gets a lucky shot before the player can zero in on the computer's cannon, then it may just win. But, given that the number is random, the odds will be rather low (perhaps one shot in 20).

```
player_cannon_angle = 45
player_cannon_power = random.randint(1,10)
```

The code to move the computer's shell is the same for the player's shell. I've opted to duplicate all of the code in this game for the sake of clarity. It would have been more efficient to use a list of bullets, like the way in which bullets were handled in The Tank Battle Game in the previous chapter, but The Artillery Gunner Game is a bit more slow-paced and precise, and will be a more interesting study without any Python language quirks.

Scoring a Hit

Because we are using drawn shapes without tapping into `MySprite` this time, we can't take advantage of Pygame's built-in collision detection. So, we'll have to write our own collision code. This will be pretty simple—we'll just use the distance function to find out the distance from one shell to one artillery gun. Let's write that function now and put it in the MyLibrary.py file:

```
# calculates distance between two points
def distance(point1, point2):
    delta_x = point1.x - point2.x
    delta_y = point1.y - point2.y
    dist = math.sqrt(delta_x*delta_x + delta_y*delta_y)
    return dist
```

If the shell is in flight, we can now use the `distance()` function to check for a collision. A distance factor of 30 should suffice, since that is the size of the artillery cannons. This is the last requirement for the game, which is now finished! Figure 13.11 shows the final result.

```
if player_firing:
    dist = distance(player_shell_position, computer_cannon_position)
    if dist < 30:
        player_score += 1
        player_firing = False

if computer_firing:
    dist = distance(computer_shell_position, player_cannon_position)
    if dist < 30:
        computer_score += 1
        computer_firing = False
```

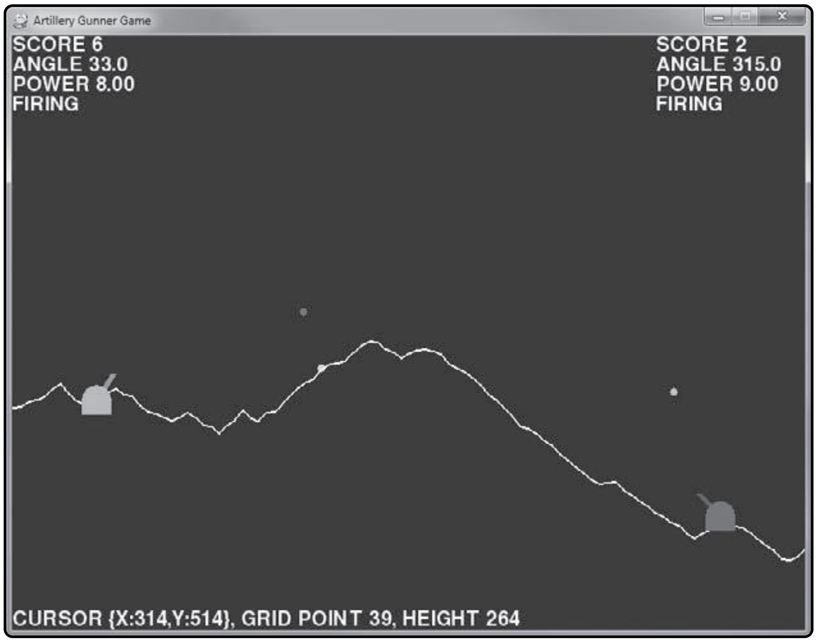


Figure 13.11

The final version of The Artillery Gunner Game!

The Complete Game

We've been looking at a lot of different sections of code for the game up to this point in the chapter. Not only that, but we built the game from scratch and evolved the code along the way. Because there have been too many changes to keep track of in a step-by-step basis, we'll include the final code listing for the game here for your perusal. The only asset requirements are the two audio files (the same boom.wav and shoot.wav files used in the previous chapter). Be sure to include the MyLibrary.py file in the folder as the game file in order for it to work.

```
# Artillery Gunner Game
# Chapter 12
import sys, time, random, math, pygame
from pygame.locals import *
from MyLibrary import *

class Terrain():
    def __init__(self, min_height, max_height, total_points):
        self.min_height = min_height
        self.max_height = max_height
        self.total_points = total_points+1
        self.grid_size = 800 / total_points
        self.height_map = list()
        self.generate()

    def generate(self):
        #clear list
        if len(self.height_map)>0:
            for n in range(self.total_points):
                self.height_map.pop()

        #first point
        last_x = 0
        last_height = (self.max_height + self.min_height) / 2
        self.height_map.append( last_height )
        direction = 1
        run_length = 0

        #remaining points
        for n in range( 1, self.total_points ):
            rand_dist = random.randint(1, 10) * direction
            height = last_height + rand_dist
            self.height_map.append( int(height) )
            if height < self.min_height: direction = -1
            elif height > self.max_height: direction = 1
            last_height = height
            if run_length <= 0:
                run_length = random.randint(1,3)
                direction = random.randint(1,2)
                if direction == 2: direction = -1
            else:
                run_length -= 1

    def get_height(self,x):
        x_point = int(x / self.grid_size)
        return self.height_map[x_point]

    def draw(self, surface):
        last_x = 0
        for n in range( 1, self.total_points ):
            #draw circle at current point
            height = 600 - self.height_map[n]
            x_pos = int(n * self.grid_size)
            pos = (x_pos, height)
            color = (255,255,255)
            if n == grid_point:
                pygame.draw.circle(surface, (0,255,0), pos, 4, 0)
            #draw line from previous point
            last_height = 600 - self.height_map[n-1]
            last_pos = (last_x, last_height)
            pygame.draw.line(surface, color, last_pos, pos, 2)
            last_x = x_pos


# this function initializes the game
def game_init():
    global screen, backbuffer, font, timer, terrain
    pygame.init()
    screen = pygame.display.set_mode((800,600))
    backbuffer = pygame.Surface((800,600))
    pygame.display.set_caption("Artillery Gunner Game")
    font = pygame.font.Font(None, 30)
    timer = pygame.time.Clock()
    #create terrain
    terrain = Terrain(50, 400, 100)


# this function initializes the audio system
def audio_init():
    global shoot_sound, boom_sound
    pygame.mixer.init()
    shoot_sound = pygame.mixer.Sound("shoot.wav")
    boom_sound = pygame.mixer.Sound("boom.wav")


# this function uses any available channel to play a sound clip
def play_sound(sound):
    channel = pygame.mixer.find_channel(True)
    channel.set_volume(0.5)
    channel.play(sound)


# these functions draw a cannon at the specified position
def draw_player_cannon(surface,position):
    #draw turret
    turret_color = (30,180,30)
    start_x = position.x + 15
    start_y = position.y + 15
    start_pos = (start_x, start_y)
    vel = angular_velocity( wrap_angle(player_cannon_angle-90) )
    end_pos = (start_x + vel.x * 30, start_y + vel.y * 30)
    pygame.draw.line(surface, turret_color, start_pos, end_pos, 6)
    #draw body
    body_color = (30,220,30)
    rect = Rect(position.x, position.y+15, 30, 15)
    pygame.draw.rect(surface, body_color, rect, 0)
    pygame.draw.circle(surface, body_color, (position.x+15,position.y+15), 15, 0)

def draw_computer_cannon(surface,position):
    #draw turret
    turret_color = (180,30,30)
    start_x = position.x + 15
    start_y = position.y + 15
    start_pos = (start_x, start_y)
    vel = angular_velocity( wrap_angle(computer_cannon_angle-90) )
    end_pos = (start_x + vel.x * 30, start_y + vel.y * 30)
    pygame.draw.line(surface, turret_color, start_pos, end_pos, 6)
    #draw body
    body_color = (220,30,30)
    rect = Rect(position.x, position.y+15, 30, 15)
    pygame.draw.rect(surface, body_color, rect, 0)
    pygame.draw.circle(surface, body_color, (position.x+15,position.y+15), 15, 0)


#main program begins
game_init()
```

```
audio_init()
game_over = False
player_score = 0
enemy_score = 0
last_time = 0
mouse_x = mouse_y = 0
grid_point = 0
player_score = computer_score = 0
player_cannon_position = Point(0,0)
player_cannon_angle = 45
player_cannon_power = 8.0
computer_cannon_position = Point(0,0)
computer_cannon_angle = 315
computer_cannon_power = 8.0
player_firing = False
player_shell_position = Point(0,0)
player_shell_velocity = Point(0,0)
computer_firing = False
computer_shell_position = Point(0,0)
computer_shell_velocity = Point(0,0)

#main loop
while True:
    timer.tick(30)
    ticks = pygame.time.get_ticks()

    #event section
    for event in pygame.event.get():
        if event.type == QUIT: sys.exit()
        elif event.type == MOUSEMOTION:
            mouse_x,mouse_y = event.pos
        elif event.type == MOUSEBUTTONUP:
            terrain.generate()

    #get key states
    keys = pygame.key.get_pressed()
    if keys[K_ESCAPE]: sys.exit()
```

```
elif keys[K_UP] or keys[K_w]:
    player_cannon_angle = wrap_angle( player_cannon_angle - 1 )

elif keys[K_DOWN] or keys[K_s]:
    player_cannon_angle = wrap_angle( player_cannon_angle + 1 )

elif keys[K_RIGHT] or keys[K_d]:
    if player_cannon_power <= 10.0:
        player_cannon_power += 0.1

elif keys[K_LEFT] or keys[K_a]:
    if player_cannon_power > 0.0:
        player_cannon_power -= 0.1

if keys[K_SPACE]:
    if not player_firing:
        play_sound(shoot_sound)
        player_firing = True
        angle = wrap_angle( player_cannon_angle - 90 )
        player_shell_velocity = angular_velocity( angle )
        player_shell_velocity.x *= player_cannon_power
        player_shell_velocity.y *= player_cannon_power
        player_shell_position = player_cannon_position
        player_shell_position.x += 15
        player_shell_position.y += 15

#update section
if not game_over:
    #keep turret inside a reasonable range
    if player_cannon_angle > 180:
        if player_cannon_angle < 270: player_cannon_angle = 270
    elif player_cannon_angle <= 180:
        if player_cannon_angle > 90: player_cannon_angle = 90

    #calculate mouse position on terrain
    grid_point = int(mouse_x / terrain.grid_size)
```

```
#move player shell
if player_firing:
    player_shell_position.x += player_shell_velocity.x
    player_shell_position.y += player_shell_velocity.y

    #has shell hit terrain?
    height = 600 - terrain.get_height(player_shell_position.x)
    if player_shell_position.y > height:
        player_firing = False

    if player_shell_velocity.y < 10.0:
        player_shell_velocity.y += 0.1

    #has shell gone off the screen?
    if player_shell_position.x < 0 or player_shell_position.x > 800:
        player_firing = False
    if player_shell_position.y < 0 or player_shell_position.y > 600:
        player_firing = False

#move computer shell
if computer_firing:
    computer_shell_position.x += computer_shell_velocity.x
    computer_shell_position.y += computer_shell_velocity.y

    #has shell hit terrain?
    height = 600 - terrain.get_height(computer_shell_position.x)
    if computer_shell_position.y > height:
        computer_firing = False

    if computer_shell_velocity.y < 10.0:
        computer_shell_velocity.y += 0.1

    #has shell gone off the screen?
    if computer_shell_position.x < 0 or computer_shell_position.x > 800:
        computer_firing = False
    if computer_shell_position.y < 0 or computer_shell_position.y > 600:
        computer_firing = False
else:
```

```
            #is the computer ready to fire?
            play_sound(shoot_sound)
            computer_firing = True
            computer_cannon_power = random.randint(1,10)
            angle = wrap_angle( computer_cannon_angle - 90 )
            computer_shell_velocity = angular_velocity( angle )
            computer_shell_velocity.x *= computer_cannon_power
            computer_shell_velocity.y *= computer_cannon_power
            computer_shell_position = computer_cannon_position
            computer_shell_position.x += 15
            computer_shell_position.y += 15

        #look for a hit by player's shell
        if player_firing:
            dist = distance(player_shell_position, computer_cannon_position)
            if dist < 30:
                play_sound(boom_sound)
                player_score += 1
                player_firing = False

        #look for a hit by computer's shell
        if computer_firing:
            dist = distance(computer_shell_position, player_cannon_position)
            if dist < 30:
                play_sound(boom_sound)
                computer_score += 1
                computer_firing = False


    #drawing section
    backbuffer.fill((20,20,120))

    #draw the terrain
    terrain.draw(backbuffer)

    #draw player's gun
    y = 600 - terrain.get_height(70+15) - 20
    player_cannon_position = Point(70,y)
```

```
draw_player_cannon(backbuffer, player_cannon_position)

#draw computer's gun
y = 600 - terrain.get_height(700+15) - 20
computer_cannon_position = Point(700,y)
draw_computer_cannon(backbuffer, computer_cannon_position)

#draw player's shell
if player_firing:
    x = int(player_shell_position.x)
    y = int(player_shell_position.y)
    pygame.draw.circle(backbuffer, (20,230,20), (x,y), 4, 0)

#draw computer's shell
if computer_firing:
    x = int(computer_shell_position.x)
    y = int(computer_shell_position.y)
    pygame.draw.circle(backbuffer, (230,20,20), (x,y), 4, 0)

#draw the back buffer
screen.blit(backbuffer, (0,0))

if not game_over:
    print_text(font, 0, 0, "SCORE " + str(player_score))
    print_text(font, 0, 20, "ANGLE " + "{:.1f}".format(player_cannon_angle))
    print_text(font, 0, 40, "POWER " + "{:.2f}".format(player_cannon_power))
    if player_firing:
        print_text(font, 0, 60, "FIRING")
    print_text(font, 650, 0, "SCORE " + str(computer_score))
    print_text(font, 650, 20, "ANGLE " + "{:.1f}".format(computer_cannon_angle))
    print_text(font, 650, 40, "POWER " + "{:.2f}".format(computer_cannon_power))
    if computer_firing:
        print_text(font, 650, 60, "FIRING")

    print_text(font, 0, 580, "CURSOR " + str(Point(mouse_x,mouse_y)) + \
        ", GRID POINT " + str(grid_point) + ", HEIGHT " + \
        str(terrain.get_height(mouse_x)))
else:
    print_text(font, 0, 0, "GAME OVER")

pygame.display.update()
```

Summary

That concludes The Artillery Gunner Game. I hope you learned a lot and enjoyed your journey through Python and Pygame. It has been a struggle at times, but a lot of fun overall. Let's see what you can do with it now!

Challenges

1. **Oh, the possibilities for this game are truly *endless*! First, it's pretty obvious that the game could use a little polish, like removing the debug messages, removing the height map terrain "crawler," and so forth. See if you can add some finishing touches to the game.**
2. **The one downer about the game is that nothing really dramatic happens when you score a hit, other than getting one point. So, when either the player or computer scores a hit, add some dramatic flair, like cycling the background color or drawing an explosion!**
3. **I would really like to see more done with the terrain's height map system. So, when a shell hits the ground, create a crater! Here's a hint: Look at the `Terrain.get_height()` method for ideas.**

CHAPTER 14

More of Everything: The Dungeon Role-Playing Game

In this chapter, we draw upon the lessons learned in all of the previous chapters to create our own dungeon role-playing game (RPG). We will pay particular attention to advanced List programming to manage the data for the game. You will gain a lot of good experience by building your own RPG based on the example presented in this chapter. But, unlike most modern RPGs that feature realistic graphics and animation, our RPG will be an *homage* to games of the past, to the early days of computing when creative storytellers had to use text to describe a fictional world. As you will soon learn, there was an offshoot of the RPG genre way back when that used text characters to represent the walls, floor, items, monsters, treasure, and even the player. These text characters were identified by their "ASCII" character codes—American Standard Code for Information Interchange. Thus, such games came to be known as "ASCII Dungeons." The sort of game we will be creating uses a random dungeon generator, but you can use the concepts presented to custom design your own game levels.

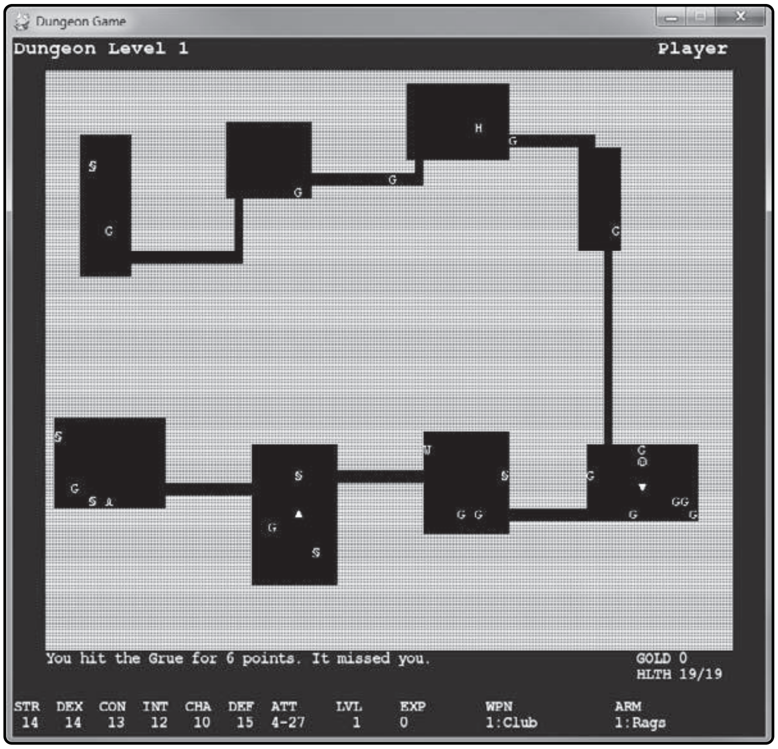
In this chapter, you will learn how to:

- Generate random dungeon rooms
- Connect the dungeon rooms with hallways
- Add gold, weapons, armor, and health potions
- Add roaming monsters that the player can fight

- Roll random character stats for the player and monsters
- Fight monsters with real to-hit, attack, and defense values
- Use your imagination—because this is an *ASCII Text RPG!*

Examining The Dungeon Game

The Dungeon RPG is shown in Figure 14.1. You will learn to build this game from scratch in this chapter. Along the way, you will learn many tricks and techniques in Python and Pygame with advanced lists and classes. And, it is rather fun to play as well!



```
Dungeon Game
Dungeon Level 1                                        Player

   You hit the Grue for 6 points. It missed you.          GOLD 0
                                                          HLTH 19/19
STR  DEX  CON  INT  CHA  DEF  ATT    LVL   EXP    WPN      ARM
 14   14   13   12   10   15  4-27     1   0      1:Club   1:Rags
```

Figure 14.1

The Dungeon RPG is a *Roguelike* game we will create in this chapter.

Review of Classic Dungeon RPGs

Today's premiere RPGs, like *Diablo III* and *World of Warcraft* (with its many expansions), would have been unbelievable to developers and gamers in the early 1980s, at the dawn of personal computers. But technology did not hold imaginative storytellers back; they still wanted to create worlds in the computer for players to have fun exploring, despite the technology. At the time, text-based displays were not considered *bad* technology, or even crude. That is simply what was available. At the time, game developers (who were really just hobbyists) were enthralled by what was possible with a computer! They didn't bemoan the lack of graphics, because there was no such thing at the time. Let's take a look at some classic examples of the genre as we plan to create one of our own design. The individual *Roguelike* games presented below, as well as *Rogue* itself, do not necessarily represent the *best* games of the genre—these are the most popular games in the genre.

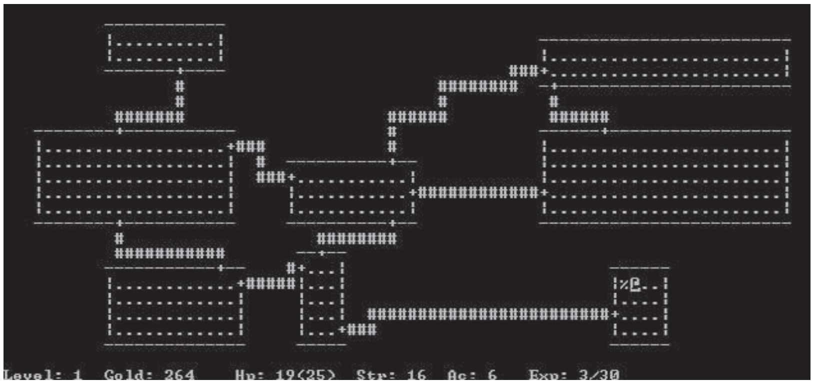
Rogue

It all started with a game called *Rogue*, according to most fans old enough to have played games in the genre in *the old days*. All games that fall into the genre are called "Roguelike," and this has become a recognized term (no space, no dash). According to Wikipedia (http://en.wikipedia.org/wiki/Roguelike), games that fit the description will have these properties:

1. Randomized game levels
2. Turn-based movement
3. *Permanent* death

The design credits for *Rogue* go to Michael Toy, Glenn Wichman, Ken Arnold, and Jon Lane.

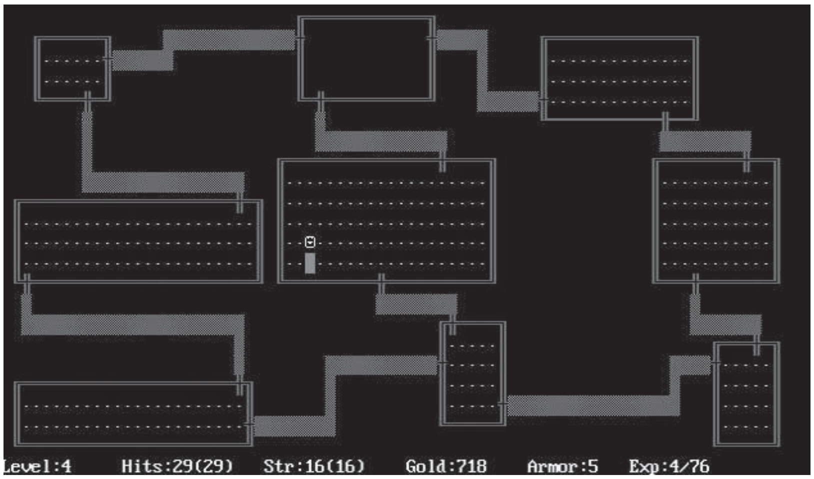
The goal of the game was to explore the dungeon down to the lowest level and retrieve a special item called the Amulet of Yendor, and then make it back up and out of the dungeon again. Even modern games like the *Diablo* series follow this basic premise, and one might even suggest that *Diablo* is a *Roguelike* game with improved graphics. But, instead of claiming treasure, your character must defeat a big bad evil boss guy in the last level. Figure 14.2 shows what the game looked like running on a Unix system.



```
Level: 1  Gold: 264   Hp: 19(25)  Str: 16  Ac: 6   Exp: 3/30
```

Figure 14.2
Rogue running on a Unix terminal. Image courtesy of Wikipedia.

The gameplay of *Rogue* is repeatable. What this means is, each level is generated and filled with monsters using the same code, so every level is based on the same algorithms. The only difference between levels is the strength of the monsters, which goes up as you descend into deeper levels. Likewise, the player's hero character gains better strength, abilities, and weapons too, so gameplay remains in balance. Figure 14.3 shows the same game running on an IBM PC.



```
Level:4    Hits:29(29)   Str:16(16)    Gold:718    Armor:5   Exp:4/76
```

FIGURE 14.3

Rogue running on an IBM PC. Image courtesy of Wikipedia.

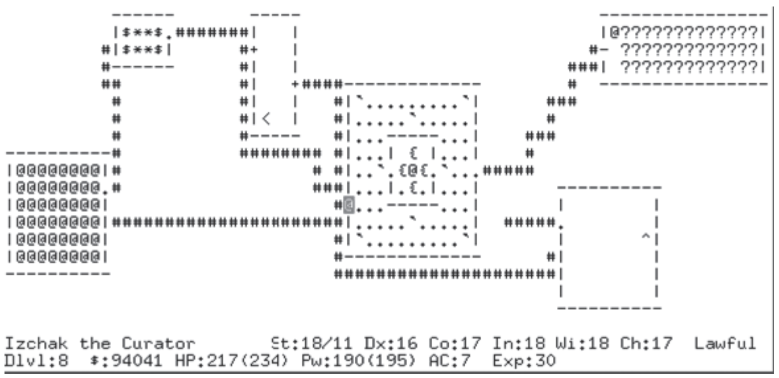
IN THE REAL WORLD

Dennis Ritchie, the creator of the C language, is reported to have said that *Rogue* wasted more CPU time than anything in history. He was referring to UNIX systems of the time.

NetHack

NetHack is an open source, freeware implementation of the gameplay found in the original *Rogue* game. Releases are maintained and available for download from http://www.nethack.org. *NetHack* is a fairly accurate version of the traditional gameplay and there are numerous variations of the game available (since it is open source). The official *NetHack* distribution includes two versions right inside the archive file when you download it. First is the usual text console version of the game with traditional ASCII characters (for purists), and this is shown in Figure 14.4.

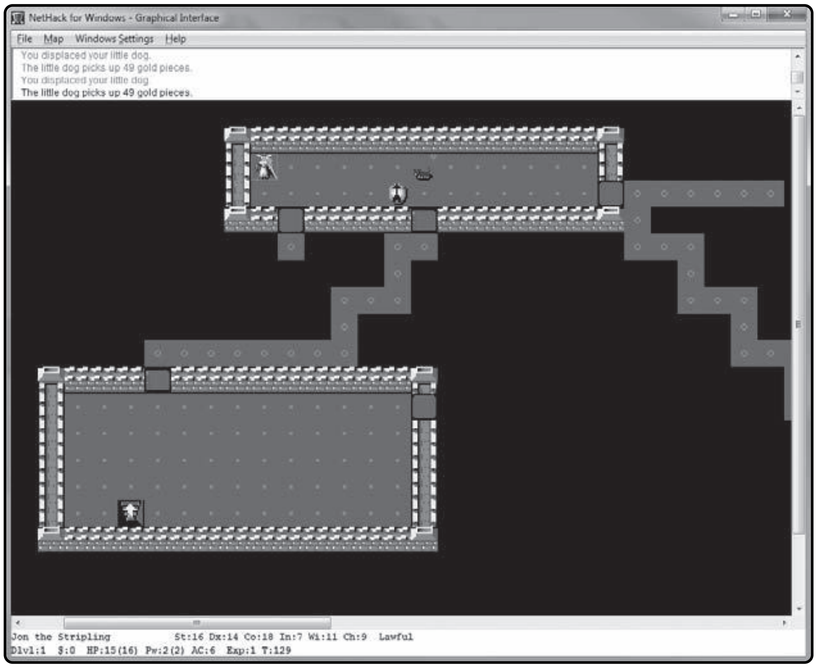
A second version, shown in Figure 14.5, is included with *NetHack* and runs in graphics mode, featuring tiled artwork. No aspect of the gameplay is changed, just the means of displaying the game.



```
Izchak the Curator       St:18/11 Dx:16 Co:17 In:18 Wi:18 Ch:17  Lawful
Dlvl:8  $:94041 HP:217(234) Pw:190(195) AC:7  Exp:30
```

FIGURE 14.4

NetHack running in a text-mode console.



```
You displaced your little dog.
The little dog picks up 49 gold pieces.
You displaced your little dog
The little dog picks up 49 gold pieces.
```

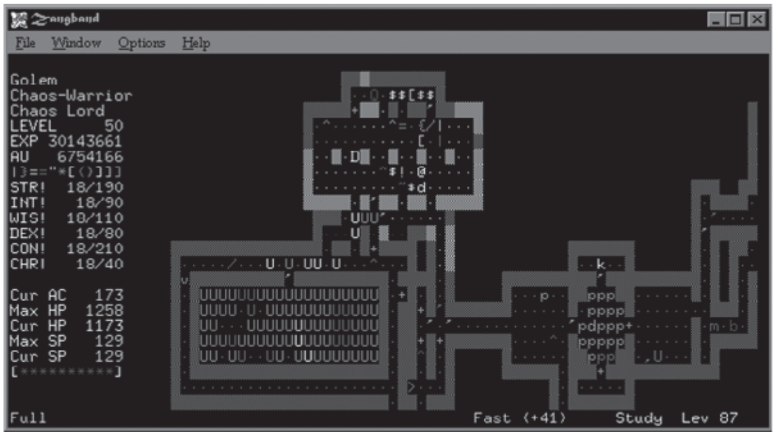
```
Jon the Stripling          St:16 Dx:14 Co:18 In:7 Wi:11 Ch:9  Lawful
Dlvl:1  $:0  HP:15(16) Pw:2(2) AC:6  Exp:1 T:129
```

FIGURE 14.5

NetHack running in graphics mode with tiled artwork.

AngBand

AngBand is another good example of the genre with very familiar gameplay and often more complex levels. This game doesn't follow the *Rogue* recipe to the last ingredient; it goes beyond to set itself apart from more traditional games. You can download the game for free from the website http://rephial.org. Figure 14.6 shows *AngBand* in action.

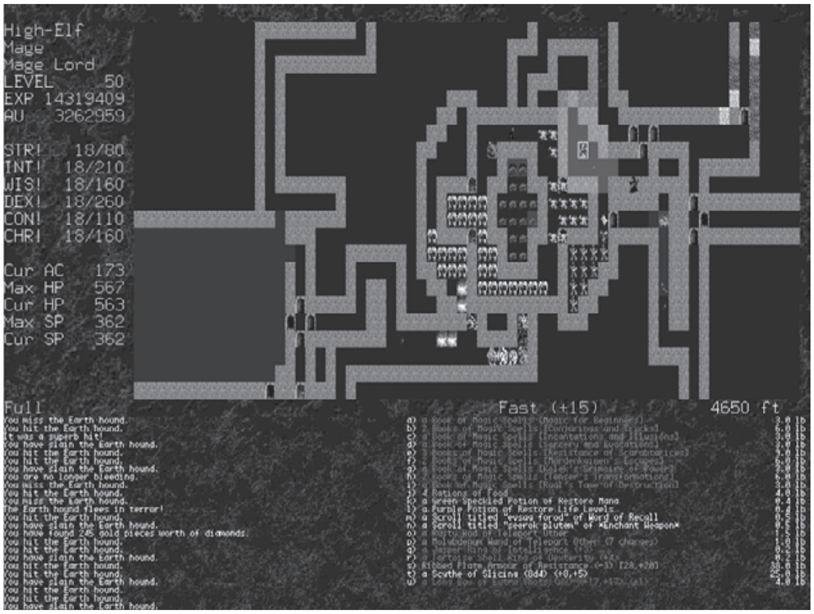


```
Golem
Chaos-Warrior
Chaos Lord
LEVEL      50
EXP 30143661
AU    6754166
STR!   18/190
INT!    18/90
WIS!   18/110
DEX!    18/80
CON!   18/210
CHR!    18/40

Cur AC    173
Max HP   1258
Cur HP   1173
Max SP    129
Cur SP    129

Full                                    Fast (+41)      Study  Lev 87
```

FIGURE 14.6

AngBand features an attractive character set sporting many colors.

Like many others in the genre, including *NetHack*, this game features two versions (catering to the two types of fans of this genre), a console-mode version and a graphics-mode version. The graphical version of *AngBand* is shown in Figure 14.7. The tiled artwork is not a huge departure from the ANSI text version, but it is enough to make the gameplay perhaps a bit more intriguing than mere text.



```
High-Elf
Mage
Mage Lord
LEVEL      50
EXP 14319409
AU    3262959

STR!   18/80
INT!  18/210
WIS!  18/160
DEX!  18/260
CON!  18/110
CHR!  18/160

Cur AC   173
Max HP   567
Cur HP   563
Max SP   362
Cur SP   362

Full                            Fast (+15)          4650 ft
You miss the Earth hound.
You hit the Earth hound.
It was a superb hit!
You have slain the Earth hound.
You hit the Earth hound.
You hit the Earth hound.
You have slain the Earth hound.
You are no longer bleeding.
You miss the Earth hound.
You hit the Earth hound.
You miss the Earth hound.
The Earth hound flees in terror!
You hit the Earth hound.
You have slain the Earth hound.
You have found 245 gold pieces worth of diamonds.
You hit the Earth hound.
You hit the Earth hound.
You have slain the Earth hound.
You hit the Earth hound.
You hit the Earth hound.
You have slain the Earth hound.
You hit the Earth hound.
You hit the Earth hound.
You have slain the Earth hound.

a) a Book of Magic Spells [Magic for Beginners]                  3.0 lb
b) 2 Books of Magic Spells [Conjurings and Tricks]               6.0 lb
c) a Book of Magic Spells [Incantations and Illusions]           3.0 lb
d) a Book of Magic Spells [Sorcery and Evocations]               3.0 lb
e) 3 Books of Magic Spells [Resistance of Scarabtarices]         9.0 lb
f) 2 Books of Magic Spells [Mordenkainen's Escapes]              6.0 lb
g) a Book of Magic Spells [Kelek's Grimoire of Power]            3.0 lb
h) 2 Books of Magic Spells [Tenser's Transformations]            6.0 lb
i) a Book of Magic Spells [Raal's Tome of Destruction]           3.0 lb
j) 4 Rations of Food                                             4.0 lb
k) a Green Speckled Potion of Restore Mana                       0.4 lb
l) a Purple Potion of Restore Life Levels                        0.4 lb
m) a Scroll titled "evsuo forod" of Word of Recall               0.5 lb
n) a Scroll titled "seerok plutem" of *Enchant Weapon*           0.5 lb
o) a Rusty Rod of Teleport Other                                 1.5 lb
p) a Molybdenum Wand of Teleport Other (7 charges)               1.0 lb
q) a Jasper Ring of Intelligence (+3)                            0.2 lb
r) a Tortoise Shell Ring of Dexterity (+4)                       0.2 lb
s) Ribbed Plate Armour of Resistance (-3) [28,+20]              38.0 lb
t) a Scythe of Slicing (8d4) (+8,+5)                            25.0 lb
u) a Long Bow of Extra Shots (x3) (+17,+17) (x1)                 4.0 lb
```

FIGURE 14.7

AngBand running in graphics mode with tiled artwork.

Sword of Fargoal

Going back a few years, we find *Sword of Fargoal* on Commodore 64, published commercially by EPYX, a popular game publishing company in the 1980s. But this game did not originate on the C=64; it was ported from an earlier Commodore PET game called *Dungeon*. It was clearly a derivative game, but the level generator used a slightly different algorithm than the one in *Rogue*. See Figure 14.8.

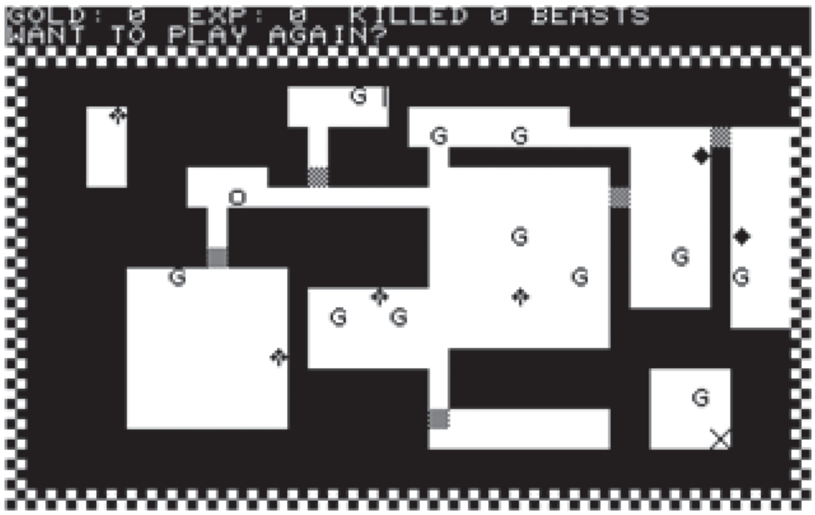


Figure 14.8
Sword of Fargoal was derived from this early Commodore PET game.

Kingdom of Kroz

Kingdom of Kroz, shown in Figure 14.9, was another classic of the 1980s. Like the similarly early versions of *NetHack*, this game employed ANSI characters to display limited animation and colored text. *Kroz* had quite complex levels because they were custom designed, not randomly generated. Believe it or not, this funny-looking screen offered a huge amount of gameplay to gamers of the time.

ZZT

ZZT was another (perhaps the best) ANSI-based *Roguelike* game. Figure 14.10 shows one screen of a much larger level. *ZZT* supported a lot of advanced gameplay features like portals and player persistence. It was developed by Tim Sweeney, founder of Epic Games, and this was the company's first game. You might recognize the name, for today Epic is responsible for the awesome *Unreal Engine 3* that powers many commercial games on Windows, Xbox 360, Linux, Mac OS X, and Sony PlayStation 3. It is cutting edge.

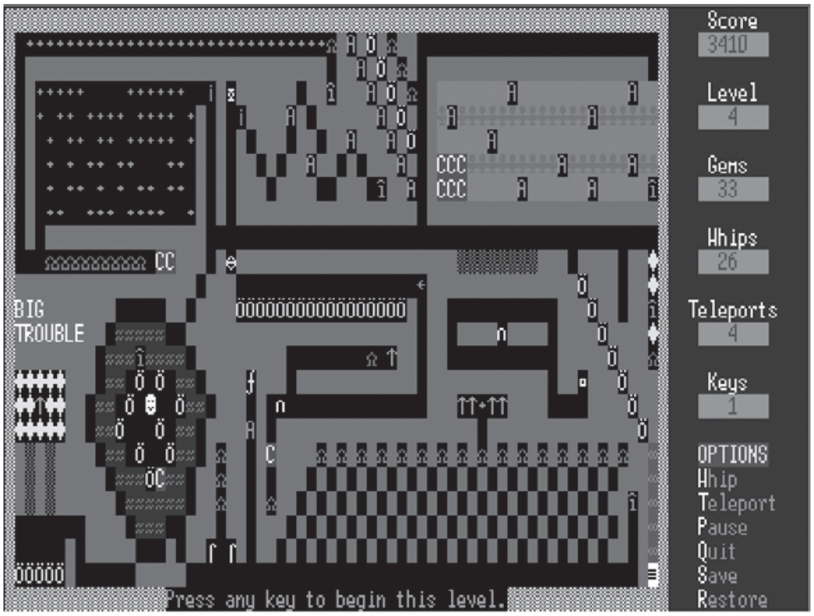


FIGURE 14.9

Kingdom of Kroz.

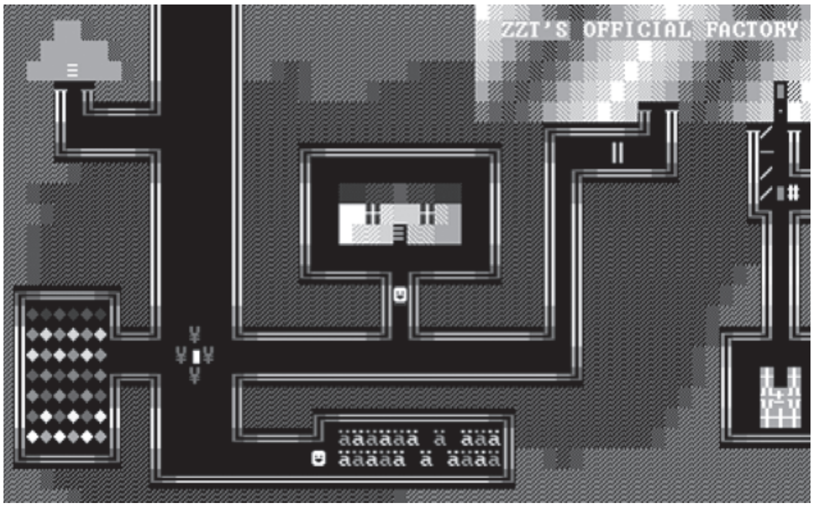


FIGURE 14.10

ZZT was created by Epic Games (of modern *Unreal Engine* fame).

Creating a Dungeon Level

The key to making a game of this type is setting up an array (or `List` in Python) to represent one game level. That array of data is recycled for each level, which is generated randomly. As the player reaches the stairs or portal to the next level down, the game should generate a new random level using the current level number with a common random number seed. This *seed* makes it possible for the game to re-generate the same levels that the player has already gone through, while travelling back *up* to exit out of the dungeon (after finding the Amulet of Yendor, if we're following the classic plot). We are going to cover enough information about the genre to build a level generator, add user input, rudimentary combat with monsters, and collision detection with the walls. But, the rest of the gameplay will be *up to you!*

Understanding ASCII Characters

When creating a *Roguelike* game in graphics mode using a library like Pygame, assuming we want the game to look and feel authentic, we have to simulate the text display. This affords us the benefit of making the game *look* authentic, but giving us the ability to do anything we want with graphics. What is a character set? It's a numbered list of characters. The standard encoding for characters is called ASCII.

ASCII Character Set

A single character has an "ASCII code," and there are 256 characters total. You might have noticed that a typical PC keyboard has only about 100 keys. That's true. The ASCII table includes some special characters used for drawing boxes on the old console displays back in the 1970s and 1980s.

HINT

The correct pronunciation of *ASCII* is "Ask-ee."

The ASCII codes will be treated like animation frames, and the character set like a sprite sheet. Each character will be handled like one frame of a large animation set. The first 31 characters on the line below were known as *non-printing characters* because they were not printed on the old console displays. These first 31 characters were used for special codes that would do things on the console display. For instance, ASCII code 10 is the linefeed character, while ASCII code 13 is the newline character. When these codes are "printed out," they perform an action rather than display a character. Incidentally, the last character on this first line is ASCII code 32, which is Space, the first printable character. The first two characters, which look like little happy faces, were often used for the player character in *Roguelike* games.

```
☺ ☻ ♥ ♦ ♣ ♠ • ◘ ○ ◙ ♂ ♀ ♪ ♫ ☼ ► ◄ ↕ ‼ ¶ § ▬ ↨ ↑ ↓ → ← ∟ ↔ ▲ ▼
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
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 [ \ ] ^ _ `
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 { | } ~ ⌂ Ç
ü é â ä à å ç ê ë è ï î ì Ä Å É æ Æ ô ö ò û ù ÿ Ö Ü ¢ £ ¥ ₧ ƒ á
```

í ó ú ñ Ñ ª º ¿ ⌐ ¬ ½ ¼ ¡ « » ░ ▒ ▓ │ ┤ ╡ ╢ ╖ ╕ ╣ ║ ╗ ╝ ╜ ╛ ┐ └
┴ ┬ ├ ─ ┼ ╞ ╟ ╚ ╔ ╩ ╦ ╠ ═ ╬ ╧ ╨ ╤ ╥ ╙ ╘ ╒ ╓ ╫ ╪ ┘ ┌ █ ▄ ▌ ▐ ▀ α
ß Γ π Σ σ µ τ Φ Θ Ω δ ∞ φ ε ∩ ≡ ± ≥ ≤ ⌠ ⌡ ÷ ≈ ° ∙ · √ ⁿ ² ■

Printing ASCII Characters

In Python, we can print ASCII characters to the console or in graphics mode with Pygame using the same font printing functions we've been using all along. But, you may be wondering, where to you *get* the ASCII characters that you want to print, if they aren't all represented on the keyboard? There are three ways to do this.

First, you can find an ASCII table (such as from a website) and copy and paste the characters into your program, as a string. For instance, to print the [PI] character to the console, copy the [PI] character out of an ASCII table and paste it into your `print()` function call, like so:

```
print("Pi looks like this: ")
```

The second way is similar, but doesn't require copying and pasting the character. Instead, we just embed the character into the string with an Alt key sequence. This is a bit of a hacker trick that most PC users don't know about anymore today, because command prompts and shells are not commonly used today. What you do is, hold down the Alt key, and type in the ASCII code using the numeric keypad. You need to know the ASCII code, of course, but we'll solve that little problem here shortly. This doesn't work with the Python IDLE editor or a Python prompt, so you'll have to use a text editor like Notepad. Try Alt+100 to see what is displayed.

But, both of those are clunky ways to print ASCII characters. The third, and preferred, way is to use code to convert an ASCII code into a character. Python has just a function called `chr()`. You have to know the ASCII code of the character you want to print, so keep an ASCII table handy. Figure 14.11 is one such table created with the following Python code.

```
print("ASCII code 100 = " + chr(100))
ASCII code 100 = d
```

The ASCII Table Program

Let's write a short program to generate an ASCII table that can be used as a reference. Remember, the Python console will not display many of escape sequence, non-printing characters. This program aligns the ASCII table into eight columns

```
cols = 8
rows = 256//cols
table = list("" for n in range(rows+1))
char = 1
```

```
#create strings filled with table data
for col in range(1,cols+1):
    for row in range(1,rows+1):
        table[row] += '{:3.0f}'.format(char) + ' '
        if char not in (9,10,13): #skip movement chars
            table[row] += chr(char)
        table[row] += '\t'
        char += 1

#print the ASCII table
for row in table:  print(row)
```

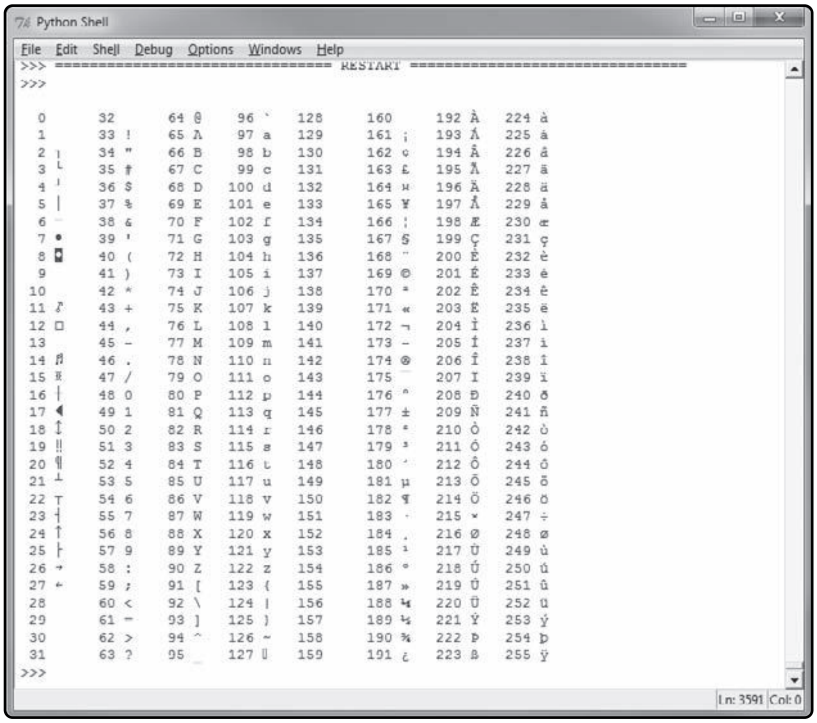
TRICK

Want to play a prank on your friends? Create a password using a common word that's easy to guess, but insert ASCII code 255 into the password. That's *another* space character!

This short Python program produces the output shown in Figure 14.11. Many of the characters will not show up in the Python Shell output window. The console was simply not designed to handle escape sequence characters. For our purposes, of using the ASCII chart as a sprite sheet, we don't need to be concerned with the original purpose of those codes. There's another more complicated problem with the console's output—the default encoding is most likely *Unicode*. So, only the first 128 characters (0 to 127) in the ASCII character set can be printed normally, while the extended codes (up to 255) will be encoded with Unicode characters. Suffice it to say, our little program has issues with character encoding.

An Improved ASCII Table Program

Here's one way to resolve the problem. Rather than using lookup code (with the `chr()` function), we can store the ASCII character set as a string and index into it with the ASCII code representing an index. This is kind of a cheat but it produces the nice-looking table we want to use as a reference. The code for formatting the characters into a table is the same, but the call to `chr()` has been replaced with an index into the `chars` string with `chars[char]`. The result is shown in Figure 14.12, and it looks just right! However, you may not be able to type in this code, especially into the IDLE editor, due to the character codes. So, please open up the source code file *ASCII Table 2.py* from the chapter resources in order to run the program.



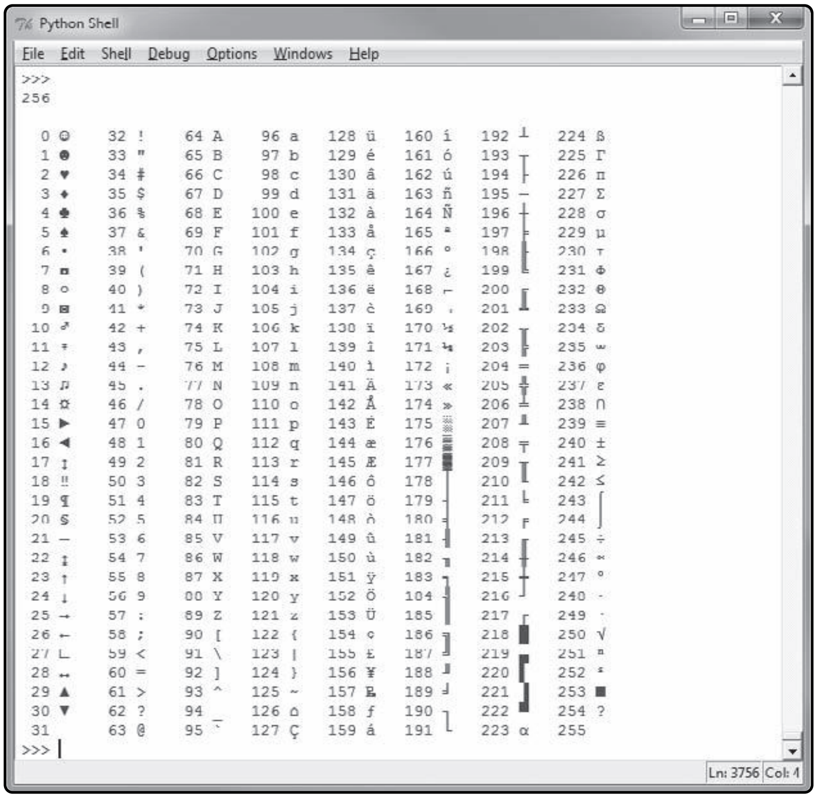
```
>>> ================================ RESTART ================================
>>>

  0         32        64 @      96 `    128       160       192 À     224 à
  1         33 !      65 A      97 a    129       161 ¡     193 Á     225 á
  2 ¦       34 "      66 B      98 b    130       162 ¢     194 Â     226 â
  3 └       35 #      67 C      99 c    131       163 £     195 Ã     227 ã
  4 ╵       36 $      68 D     100 d    132       164 ¤     196 Ä     228 ä
  5 │       37 %      69 E     101 e    133       165 ¥     197 Å     229 å
  6 ─       38 &      70 F     102 f    134       166 ¦     198 Æ     230 æ
  7 •       39 '      71 G     103 g    135       167 §     199 Ç     231 ç
  8 ◘       40 (      72 H     104 h    136       168 ¨     200 È     232 è
  9         41 )      73 I     105 i    137       169 ©     201 É     233 é
 10         42 *      74 J     106 j    138       170 ª     202 Ê     234 ê
 11 ♂       43 +      75 K     107 k    139       171 «     203 Ë     235 ë
 12 □       44 ,      76 L     108 l    140       172 ¬     204 Ì     236 ì
 13         45 -      77 M     109 m    141       173 -     205 Í     237 í
 14 ♫       46 .      78 N     110 n    142       174 ®     206 Î     238 î
 15 ☼       47 /      79 O     111 o    143       175 ¯     207 Ï     239 ï
 16 │       48 0      80 P     112 p    144       176 °     208 Ð     240 ð
 17 ◄       49 1      81 Q     113 q    145       177 ±     209 Ñ     241 ñ
 18 ↕       50 2      82 R     114 r    146       178 ²     210 Ò     242 ò
 19 ‼       51 3      83 S     115 s    147       179 ³     211 Ó     243 ó
 20 ¶       52 4      84 T     116 t    148       180 ´     212 Ô     244 ô
 21 ┴       53 5      85 U     117 u    149       181 µ     213 Õ     245 õ
 22 ┬       54 6      86 V     118 v    150       182 ¶     214 Ö     246 ö
 23 ┤       55 7      87 W     119 w    151       183 ·     215 ×     247 ÷
 24 ↑       56 8      88 X     120 x    152       184 ¸     216 Ø     248 ø
 25 ├       57 9      89 Y     121 y    153       185 ¹     217 Ù     249 ù
 26 →       58 :      90 Z     122 z    154       186 º     218 Ú     250 ú
 27 ←       59 ;      91 [     123 {    155       187 »     219 Û     251 û
 28         60 <      92 \     124 |    156       188 ¼     220 Ü     252 ü
 29         61 =      93 ]     125 }    157       189 ½     221 Ý     253 ý
 30         62 >      94 ^     126 ~    158       190 ¾     222 Þ     254 þ
 31         63 ?      95 _     127 ⌂    159       191 ¿     223 ß     255 ÿ
>>>
```

FIGURE 14.11

Output from the ASCII Table program.

```
chars = \
"☺ ☻♥♦♣♠•◘○◙♂♀♪♫☼► ◄↕‼¶§▬↨↑↓→←∟↔▲▼·!\"#$%&'()*+,-./0123456789:;<=>?@"\
"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~⌂Ç"\
"üéâäàåçêëèïîìÄÅÉæÆôöòûùÿÖÜ¢£¥₧ƒáíóúñÑªº¿⌐¬½¼¡«»░▒▓│┤╡╢╖╕╣║╗╝╜╛┐ └"\
"┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘ ┌█▄▌▐▀αßΓπΣσµτΦΘΩδ∞φε∩≡±≥≤⌠⌡÷≈°··√ⁿ²■?º"

cols = 8
rows = 256//cols
table = list("" for n in range(rows+1))
char = 0

for col in range(1,cols+1):
    for row in range(1,rows+1):
        table[row] += '{:3.0f}'.format(char) + ' '
        table[row] += chars[char]
        table[row] += '\t'
        char += 1
```

```
print(len(chars))
for row in table: print(row)
```



```
>>>
256

  0 ☺     32 !     64 A     96 a    128 ü    160 í    192 ┴    224 ß
  1 ☻     33 "     65 B     97 b    129 é    161 ó    193 ┬    225 Γ
  2 ♥     34 #     66 C     98 c    130 â    162 ú    194 ├    226 π
  3 ♦     35 $     67 D     99 d    131 ä    163 ñ    195 ─    227 Σ
  4 ♣     36 %     68 E    100 e    132 à    164 Ñ    196 ┼    228 σ
  5 ♠     37 &     69 F    101 f    133 å    165 ª    197 ╞    229 µ
  6 •     38 '     70 G    102 g    134 ç    166 º    198 ╟    230 τ
  7 ◘     39 (     71 H    103 h    135 ê    167 ¿    199 ╚    231 Φ
  8 ○     40 )     72 I    104 i    136 ë    168 ⌐    200 ╔    232 Θ
  9 ◙     41 *     73 J    105 j    137 è    169 ¬    201 ╩    233 Ω
 10 ♂     42 +     74 K    106 k    138 ï    170 ½    202 ╦    234 δ
 11 ♀     43 ,     75 L    107 l    139 î    171 ¼    203 ╠    235 ∞
 12 ♪     44 -     76 M    108 m    140 ì    172 ¡    204 ═    236 φ
 13 ♫     45 .     77 N    109 n    141 Ä    173 «    205 ╬    237 ε
 14 ☼     46 /     78 O    110 o    142 Å    174 »    206 ╧    238 ∩
 15 ►     47 0     79 P    111 p    143 É    175 ░    207 ╨    239 ≡
 16 ◄     48 1     80 Q    112 q    144 æ    176 ▒    208 ╤    240 ±
 17 ↕     49 2     81 R    113 r    145 Æ    177 ▓    209 ╥    241 ≥
 18 ‼     50 3     82 S    114 s    146 ô    178 │    210 ╙    242 ≤
 19 ¶     51 4     83 T    115 t    147 ö    179 ┤    211 ╘    243 ⌠
 20 §     52 5     84 U    116 u    148 ò    180 ╡    212 ╒    244 ⌡
 21 ▬     53 6     85 V    117 v    149 û    181 ╢    213 ╓    245 ÷
 22 ↨     54 7     86 W    118 w    150 ù    182 ╖    214 ╫    246 ≈
 23 ↑     55 8     87 X    119 x    151 ÿ    183 ╕    215 ╪    247 °
 24 ↓     56 9     88 Y    120 y    152 Ö    184 ╣    216 ┘    248 ∙
 25 →     57 :     89 Z    121 z    153 Ü    185 ║    217 ┌    249 ·
 26 ←     58 ;     90 [    122 {    154 ¢    186 ╗    218 █    250 √
 27 ∟     59 <     91 \    123 |    155 £    187 ╝    219 ▄    251 ⁿ
 28 ↔     60 =     92 ]    124 }    156 ¥    188 ╜    220 ▌    252 ²
 29 ▲     61 >     93 ^    125 ~    157 ₧    189 ╛    221 ▐    253 ■
 30 ▼     62 ?     94 _    126 ⌂    158 ƒ    190 ┐    222 ▀    254 ?
 31       63 @     95 `    127 Ç    159 á    191 └    223 α    255
>>>
```

Figure 14.12
An improved ASCII Table program now produces useful output we can use as a reference.

Simulating a Text Console Display

Now that we understand the ASCII character set, we can use this information to simulate a text console display for our own *Roguelike* game. Since we're only simulating a console display, and not duplicating one perfectly, we can cheat a little on the dimensions. The old console displays that *Roguelike* games were played on could display text characters—80 across and 25 down. We'll extend it to 80 × 45 for a better ratio, since most modern LCD screens have a widescreen orientation now with a ratio of 4:3 or 16:9 or similar. In these terms, an old CRT (cathode ray tube) monitor had a ratio of 8:2.5. It worked, though, because the characters were taller than they were wide. Figure 14.13 shows an illustration of that space with the actual number of tiles we have available.

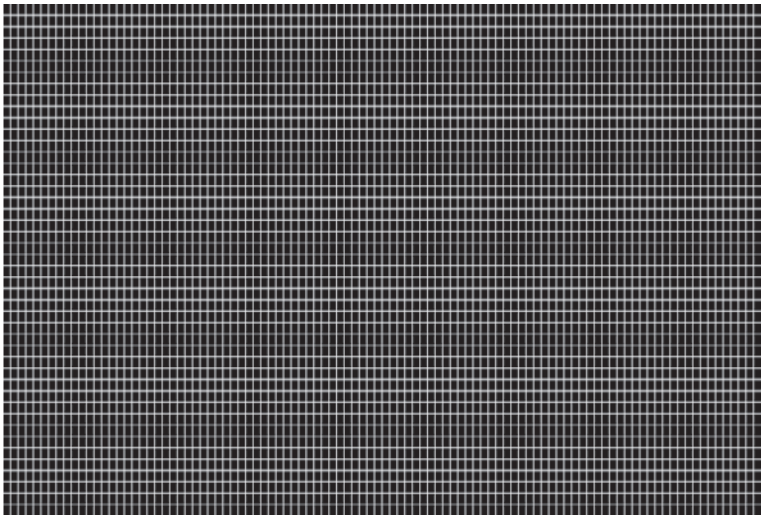


Figure 14.13

This 80 × 45 grid represents the space available for one game level.

Keeping Track of Tiles

Since each character represents one gameplay "tile," then what we have is a game level that will support 80 × 45 = 3,600 tiles. That is actually quite a lot of space, considering that the game has many such levels (down into the depths).

Now, to fill the game level, we need a pair of objects: a `list` and a `MySprite`. First, the `list` will be created and filled with a default value.

```
tiles = list()
for y in range(0,45):
    for x in range(0,80):
        tiles.append(8)
```

ASCII Table as a Sprite

Next, the `MySprite` object will be created, and a sprite sheet containing all of the ASCII characters will be loaded and treated like frames of animation. The bitmap is shown in Figure 14.14. This file is supplied with the chapter resource files, and is called ascii8x12.png.

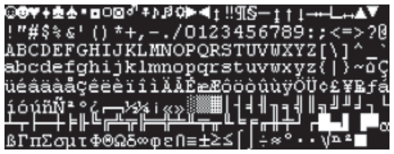


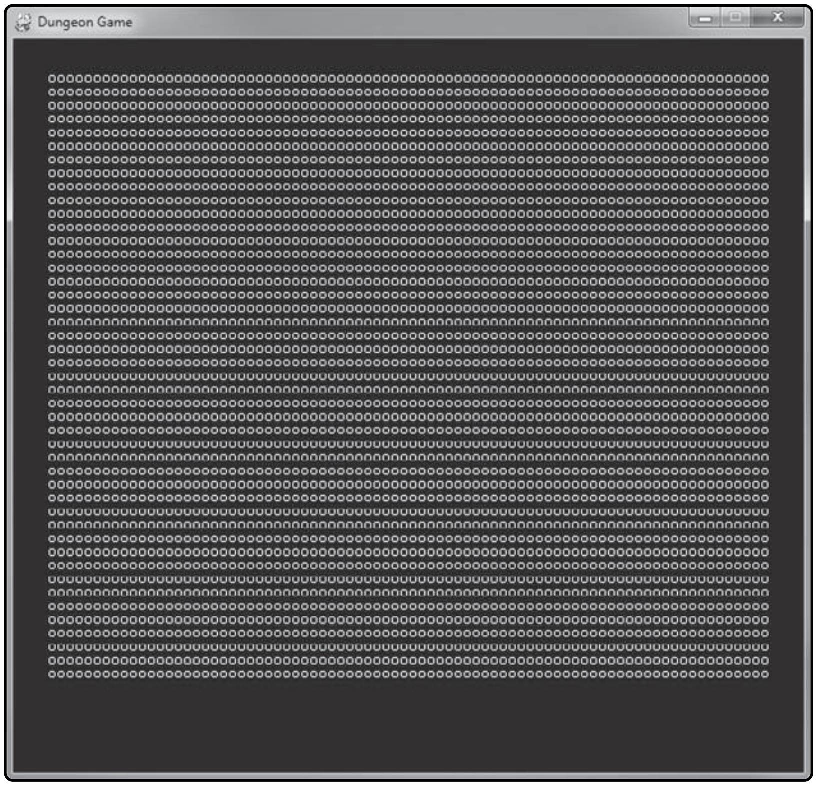
Figure 14.14

A sprite sheet containing ASCII characters.

```
text = MySprite()
text.load("ascii8x12.png", 8, 12, 32)
```

Drawing the Dungeon Level

Now we can simply draw the dungeon level by referencing the tiles list, which contains the ASCII code of each tile, and using that as the animation frame index in the sprite. Multiply this process across 80 columns and 45 rows, as the following code does, and we have a level filled with a default ASCII character as shown in Figure 14.15. At this point, the "basic mechanics" of the game is working! This was the hard part. Now, we can focus on generating a random level with rooms and passages.



```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Figure 14.15
The first trial run of The Dungeon Game with working level data.

```
for y in range(0,45):
    for x in range(0,80):
        index = y * 80 + x
        value = tiles[index]
        text.X = 30 + x * 8
        text.Y = 30 + y * 12
        text.frame = text.last_frame = value
        text.update(0)
        text.draw(surface)
```

Generating Random Rooms

A hallmark feature of *Roguelike* games is endless replay value because the game levels are randomly generated. Every time you play, the game will be different! To generate a single level, we will fill it with rooms. Now, this is where some creative programming comes in, because there are many ways to do this. In the example presented here, there are eight total rooms in one level—four above, four below. You might take a different approach and generate one large room in the middle with several smaller rooms scattered around it. There are many possibilities! Go ahead and experiment with the Python/Pygame code found here in this chapter and see what interesting new game levels you can come up with!

Creating the Dungeon Class

We have reached a complexity level that requires a class to continue further with the construction of our dungeon. Here is a class called `Dungeon` that will help organize the data and code. This class will be responsible for generating and drawing random levels. Of particular interest are the two helper methods: `getCharAt()` and `setCharAt()`. We will need these to generate a random level. Note that, as was the case in all previous chapters, the MyLibrary.py file must be found in the same folder as the game file so that classes like MySprite are available.

```
class Dungeon():
    def __init__(self):
        #create the font sprite
        self.text = MySprite()
        self.text.load("ascii8x12.png", 8, 12, 32)

        #create the level list
        self.tiles = list()
        for n in range(0,80*45):
            self.tiles.append(-1)

    def getCharAt(self, x, y):
        index = y * 80 + x
        return self.tiles[index]
    def setCharAt(self, x, y, char):
        index = y * 80 + x
        self.tiles[index] = char

    def draw(self, surface, offx, offy):
        for y in range(0,45):
            for x in range(0,80):
                value = self.getCharAt(x,y)
                if value >= 0:
                    self.text.X = offx + x * 8
                    self.text.Y = offy + y * 12
                    self.text.frame = value
                    self.text.last_frame = value
                    self.text.update(0)
                    self.text.draw(surface)
```

Now, the prototype will just generate random levels every time the Space key is pressed. This is an important step, to validate whether the level generating algorithm is free of bugs. We don't want rooms overlapping or hallways missing the mark. To generate the rooms, we will use a rectangle to represent each room, and a list will turn the rooms into an easy-to-manage array. There are three main ASCII codes that will be used to build a dungeon level:

1. Code 175, Char: '░' (background)
2. Code 177, Char: '▓' (hallways)
3. Code 218, Char: '█' (rooms)

Generating the Northern Rooms

Let's start off with the top of the level, in which we will put four rooms. Each room will have a slightly random position (within a small variance) and a slightly random size (with a minimum and maximum). After creating the rooms, the `Dungeon.generate()` method then fills in the tiles array/list with the room data.

```
def generate(self, emptyChar=175, roomChar=218, hallChar=177):
    #clear existing level
    for index in range(0,80*45):
        self.tiles[index] = emptyChar

    #create random rooms
    self.rooms = list()
```

```
PL = 4
PH = 8
SL = 5
SH = 14
room = Rect(0 + random.randint(1,PL),
            0 + random.randint(1,PH),
            random.randint(SL,SH),
            random.randint(SL,SH))
self.rooms.append(room)
room = Rect(20 + random.randint(1,PL),
            0 + random.randint(1,PH),
            random.randint(SL,SH),
            random.randint(SL,SH))
self.rooms.append(room)
room = Rect(40 + random.randint(1,PL),
            0 + random.randint(1,PH),
            random.randint(SL,SH),
            random.randint(SL,SH))
self.rooms.append(room)
room = Rect(60 + random.randint(1,PL),
            0 + random.randint(1,PH),
            random.randint(SL,SH),
            random.randint(SL,SH))
self.rooms.append(room)

#add rooms to level
for room in self.rooms:
    for y in range(room.y,room.y+room.height):
        for x in range(room.x,room.x+room.width):
            self.setCharAt(x, y, roomChar)
```

Figure 14.16 shows the result so far. We're making good progress already, and it seems to be moving along quickly now with the `Dungeon` class.

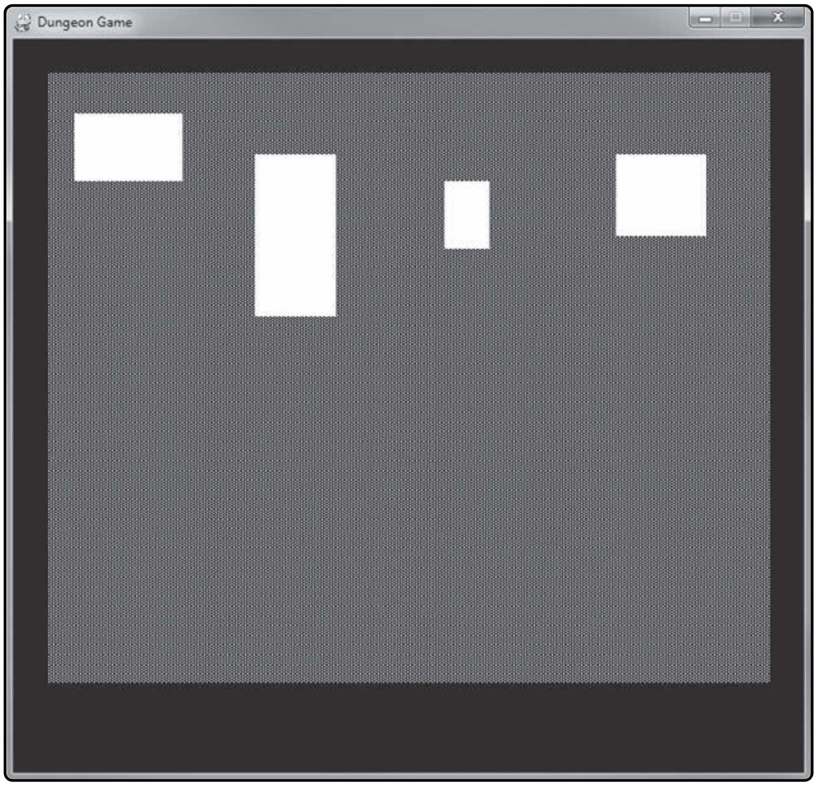


Figure 14.16
The four northern rooms have been generated.

Generating the Southern Rooms

Next, we use similar code with a few minor changes to the Y position of each to generate the four southern rooms, and end up with the level shown in Figure 14.17.

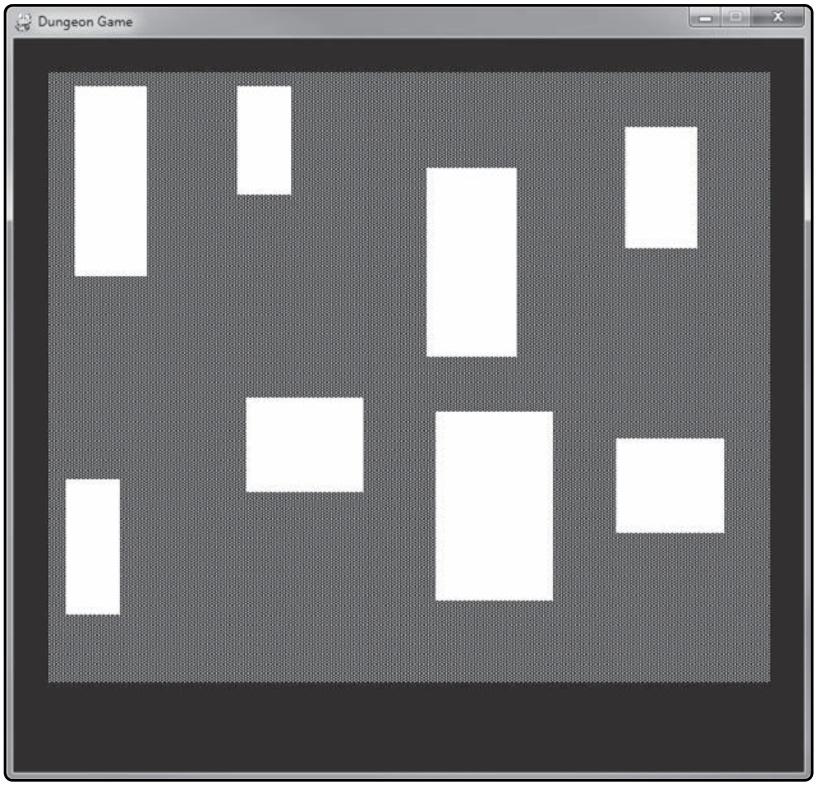


Figure 14.17

The four southern rooms have been generated.

```
room = Rect(0 + random.randint(1,PL),
            22 + random.randint(1,PH),
            random.randint(SL,SH),
            random.randint(SL,SH))
self.rooms.append(room)
room = Rect(20 + random.randint(1,PL),
            22 + random.randint(1,PH),
            random.randint(SL,SH),
            random.randint(SL,SH))
self.rooms.append(room)
room = Rect(40 + random.randint(1,PL),
            22 + random.randint(1,PH),
            random.randint(SL,SH),
            random.randint(SL,SH))
```

```
    self.rooms.append(room)
    room = Rect(60 + random.randint(1,PL),
                22 + random.randint(1,PH),
                random.randint(SL,SH),
                random.randint(SL,SH))
    self.rooms.append(room)
```

This room code is now working great! However, there's a lot of repeated code here with only minor differences in the code from one room to the next. We can take advantage of this repeatability by writing a reusable method that will work for all eight rooms. So, let's abolish the room generation code just written and replace it with calls to this new method:

```
def createRoom(self,x,y,rposx,rposy,rsizel,rsizeh):
    room = Rect(x + random.randint(1,rposx),
                y + random.randint(1,rposy),
                random.randint(rsizel,rsizeh),
                random.randint(rsizel,rsizeh))
    self.rooms.append(room)
```

Using this new helper method, the code to generate all eight rooms (within `generate()`) is much more manageable. This will also make it much easier to experiment with different level generation algorithms. This is an example of a proper method, created to eliminate or *optimize* repeating code. The variables `PL`, `PH`, `SL`, and `SH`, represent the random position and size of each room. Feel free to experiment with different values!

```
    PL = 4
    PH = 8
    SL = 5
    SH = 14
    self.rooms = list()
    self.createRoom(0,0,PL,PH,SL,SH)
    self.createRoom(20,0,PL,PH,SL,SH)
    self.createRoom(40,0,PL,PH,SL,SH)
    self.createRoom(60,0,PL,PH,SL,SH)
    self.createRoom(0,22,PL,PH,SL,SH)
    self.createRoom(20,22,PL,PH,SL,SH)
    self.createRoom(40,22,PL,PH,SL,SH)
    self.createRoom(60,22,PL,PH,SL,SH)
```

Generating Random Hallways

The hallways or passages connect the rooms, and are the key to making random levels. To generate the hallways, we will make some assumptions that other *Roguelike* algorithms might not make. In other words, there are probably more creative path-finding algorithms that link any one room to any other room in the level. But, let's try to keep it simple and connect two rooms that are near each other.

TRICK

While running the game, at any time press the Space key to re-generate the level. This is good for testing and debugging the game, but should be removed when the game is finished.

Horizontal Hallways

We'll plan our hallway code with reusability in mind first (and wisely put it into a method from the start). First, we have the source room. Pick a random location along the right edge of the room for the starting point of the hallway. Then, move the hallway toward the right, one tile at a time, until it reaches the position of the destination room. If we have hit the destination room already, then that's it, the hallway is finished. But, most likely, the hallway will need to go up or down to reach the room. So, if the position of the room is below or above, we route the hall up or down accordingly until we bump into it. Let's see how it will look first, then show the code. First, in Figure 14.18, we have the condition where the hall ran straight across into the second room without any turns needed.

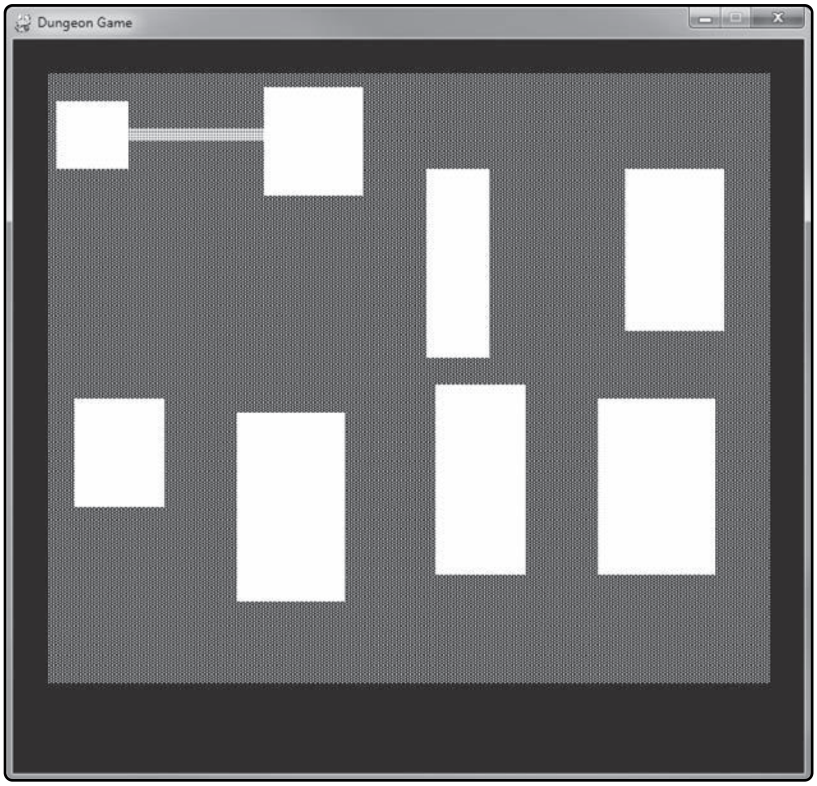


FIGURE 14.18

A straight passageway.

Next, we can show the condition where a hallway might need to angle up or down to get to the destination room, which is the situation shown in Figure 14.19. At this point, we can repeat the process between the northern four rooms, and then for the southern four rooms. Let's see the new `Dungeon.createHallRight()` method first.

HINT

Some random dungeons will look *fantastic*, while some will look horrid. It's a matter of fine-tuning the algorithm to meet your design expectations. Most likely the best modifications to make will be to the random ranges used to set the position and size of each room. The hallways usually work fine if the rooms are reasonably placed.

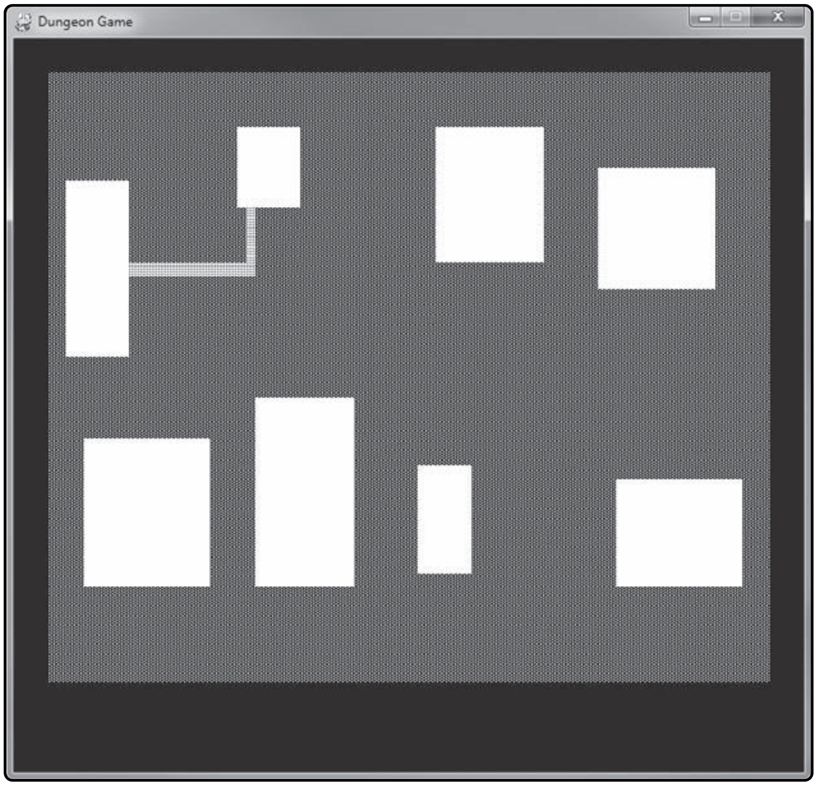


FIGURE 14.19

An angled passageway.

```
def createHallRight(self,src,dst,hallChar):
    pathx = src.x + src.width
    pathy = src.y + random.randint(1,src.height-2)
    self.setCharAt(pathx,pathy,hallChar)
    if pathy > dst.y and pathy < dst.y + dst.height:
        while pathx < dst.x:
            pathx += 1
            self.setCharAt(pathx,pathy,hallChar)
    else:
        while pathx < dst.x+1:
            pathx += 1
            self.setCharAt(pathx,pathy,hallChar)
        if pathy < dst.y+1:
            self.setCharAt(pathx,pathy,hallChar)
            while pathy < dst.y:
                pathy += 1
                self.setCharAt(pathx,pathy,hallChar)
        else:
            self.setCharAt(pathx,pathy,hallChar)
            while pathy > dst.y + dst.height:
                pathy -= 1
                self.setCharAt(pathx,pathy,hallChar)
```

Because three hallways connect four rooms, we have six total hallways to create. Note how it is a simple connection of one room to another. This could be replaced with a `for` loop but the code is more "self documenting" this way.

```
        self.createHallRight(self.rooms[0],self.rooms[1],hallChar)
        self.createHallRight(self.rooms[1],self.rooms[2],hallChar)
        self.createHallRight(self.rooms[2],self.rooms[3],hallChar)
        self.createHallRight(self.rooms[4],self.rooms[5],hallChar)
        self.createHallRight(self.rooms[5],self.rooms[6],hallChar)
        self.createHallRight(self.rooms[6],self.rooms[7],hallChar)
```

Vertical Hallways

At this point, we have the northern rooms and southern rooms connected to each other, but the north and south "wings," so to speak, are not accessible to each other. For that, we need vertical hallways as well. We don't want too many hallways, or it will be too easy to clear the level. Instead, we'll connect the north and south wings with just one hall, and we'll do it by choosing a random room so it will be different every time.

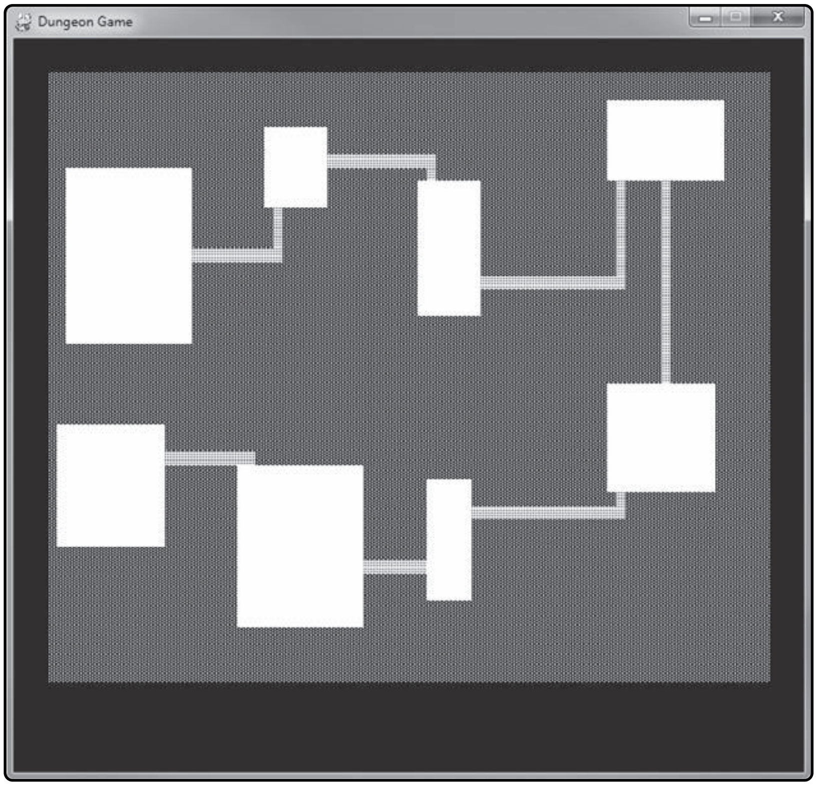


FIGURE 14.20

A hallway now connects the north and south wings.

```
def createHallDown(self,src,dst,hallChar):
    pathx = src.x + random.randint(src.width-2)
    pathy = src.y + src.height
    self.setCharAt(pathx,pathy,hallChar)
    if pathx > dst.x and pathx < dst.x + dst.width:
        while pathy < dst.y:
            pathy += 1
            self.setCharAt(pathx,pathy,hallChar)
    else:
        while pathy < dst.y+1:
            pathy += 1
            self.setCharAt(pathx,pathy,hallChar)
        if pathx < dst.x+1:
            self.setCharAt(pathx,pathy,hallChar)
            while pathx < dst.x:
                pathx += 1
                self.setCharAt(pathx,pathy,hallChar)
        else:
            self.setCharAt(pathx,pathy,hallChar)
            while pathx > dst.x + dst.width:
                pathx -= 1
                self.setCharAt(pathx,pathy,hallChar)
```

The choice of which rooms on the north to connect down south is entirely up to you. Perhaps you will connect a north room on the left with a south room on the right, and create a long, winding corridor between them? For the example, we will stick with a simple downward path that will angle left or right to reach the target room.

```
choice = random.randint(0,3)
print("choice:" + str(choice) + "," + str(choice+4))
self.createHallDown(self.rooms[choice],self.rooms[choice+4],hallChar)
```

HINT

Try connecting the rooms *vertically* with one *horizontal* hallway to invert the overall look of the dungeon! You might even make such changes with random rolls so that sometimes they just happen without code changes. Another type of dungeon that would be compelling might have one large room with small rooms arrayed around it. The important thing is making sure your algorithm creates a reasonable dungeon *every time* so the player is never given an impossible level.

Handling Range Errors

When experimenting with code such as this, with various algorithms for generating rooms, it often happens that we try to draw a room or hall outside the borders of the screen. It just happens.

```
IndexError: list assignment index out of range
```

So, it would be helpful to trap those errors when they occur rather than allowing the game to simply crash. Trapping such an error helps to diagnose the logic bug. First, we can modify `Dungeon.setCharAt()` so that it checks the ranges before trying to set a tile in the dungeon. We can also modify `Dungeon.getCharAt()`, although errors will be fewer there. This version will avoid crashing the program due to a range error. The purpose is not to let the user keep playing, but to notify *you* when a crash is about to happen, in order to fix the bug. I used this very code to solve a bug in the hall generation code and you will need it too when you experiment with new algorithms. When the game is finished, though, you can safely comment out these debugging lines.

```
def getCharAt(self, x, y):
    if x < 0 or x > 79 or y < 0 or y > 44:
        print("error: x,y = ", x, y)
        return
    index = y * 80 + x
    if index < 0 or index > 80*45:
        print("error: index = ", index)
        return
    return self.tiles[index]

def setCharAt(self, x, y, char):
    if x < 0 or x > 79 or y < 0 or y > 44:
        print("error: x,y = ", x, y)
        return
    index = y * 80 + x
    if index < 0 or index > 80*45:
        print("error: index = ", index)
        return
    self.tiles[index] = char
```

HINT

Do you prefer levels jam-packed with halls and rooms? Our generator leaves a lot of empty space because these are just the *major* rooms and halls. Using the code available, you can generate smaller off-shoot rooms and halls in any unused area. Some *Roguelike* games would even add rooms connected with *hidden* passages!

Populating the Dungeon

There are two ways to populate the dungeon: an easy way, and a hard way. The easy way involves just dropping things right into the tile array so that an ASCII character shows up in that location. The player can then interact with it based on that ASCII code. If that code represents treasure, then the player picks it up. If the code is a portal, then the player moves. If it's a monster, the player fights.

The hard way is to maintain secondary lists of objects (treasure, portals, monsters, etc.), and draw these items over the tiles of the dungeon structure, so to speak. This gives the benefit of a more attractive appearance at the cost of some very challenging code to write and maintain.

One could argue in favor of both techniques, or even suggest other alternatives. For the sake of everyone's sanity, we're going to use the easy method in this chapter. Not only is it much less difficult to work with the code, but the game will be more useful to educators who want to use this game as an experiment testbed for "bot" A.I. projects, and sticking with a list of tiles that contains everything in the game makes the code much more accessible.

Adding the Entrance and Exit Portals

The entrance portal will be a tile in one of the rooms that sends the player *up* to the previous level (or out of the dungeon if it's the first level). We want to position the entrance portal in a room so that the player starts off in that location. It would be best if the exit was not in the same room or else the player can quickly skip the whole level! First, we'll choose a random room, and then just position the portal in the middle of the room.

```
choice = random.randint(0,7)
self.entrance_x = self.rooms[choice].x + self.rooms[choice].width//2
self.entrance_y = self.rooms[choice].y + self.rooms[choice].height//2
self.setCharAt(x,y,29) #entry portal
print("entrance:",choice,x,y)
```

We use class variables for the entrance in order to keep track of it more easily for positioning the player. The exit portal will be a tile in a random location in the level that takes the player down to the next level below. The goal of the game in classic *Rogue* is to reach the last level and claim the Amulet of Yendor, then make your way back up again.

```
choice2 = random.randint(0,7)
while choice2 == choice:
    choice2 = random.randint(0,7)
x = self.rooms[choice2].x + self.rooms[choice2].width//2
y = self.rooms[choice2].y + self.rooms[choice2].height//2
self.setCharAt(x,y,30) #exit portal
print("exit:",choice2,x,y)
```

The end result of our new entrance and exit portal code is shown in Figure 14.21. The entrance portal looks like an "up" arrow, while the exit looks like a "down" arrow (ASCII codes 29 and 30, respectively). You may change these to different characters if you wish. Do you see how easy it was to add these tiles using the "easy" level array? Just set any position in the dungeon to any code from 0 to 255, and that change instantly shows up. You could even add new secret passages or cause "cave-ins."

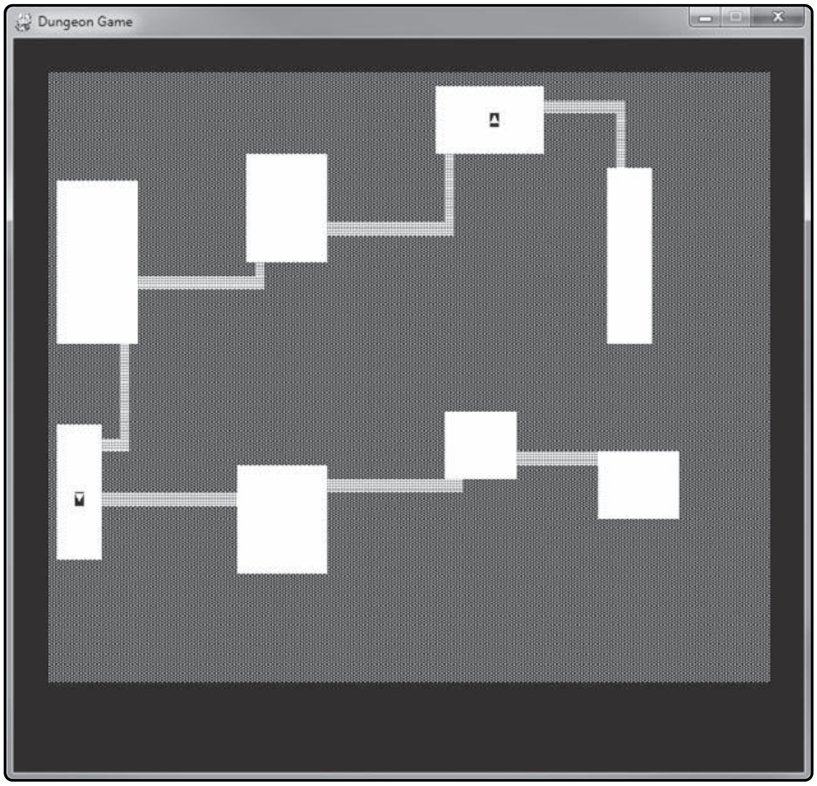


FIGURE 14.21

The entrance and exit portals.

Adding Gold

We'll follow the old-school *Roguelike* approach and use "G" to represent gold. To add random gold throughout the level, just pick a random location, check to make sure it isn't solid rock (the "background" or "empty" character is 175), which is a character with a dot pattern that makes it appear dark gray. You can change that character if you wish, but just be sure to be consistent in your code. It wouldn't be a bad idea to define constants for these characters.

But, instead of looking specifically at the empty tile code, let's look for the *room* code instead, and only drop gold into a room. This eliminates some code we would have to write to pick up gold in the halls, and centers gameplay in the rooms.

Now, let's add some random gold. First, choose a random number of drops, then scatter them around the level on valid tiles. This is where it becomes handy to have an ASCII table for reference. Figure 14.22 shows the result of the random gold scattered around the level. Isn't it remarkable how easy it is to add each new feature to the game? That is because we're building on a solid foundation of code.

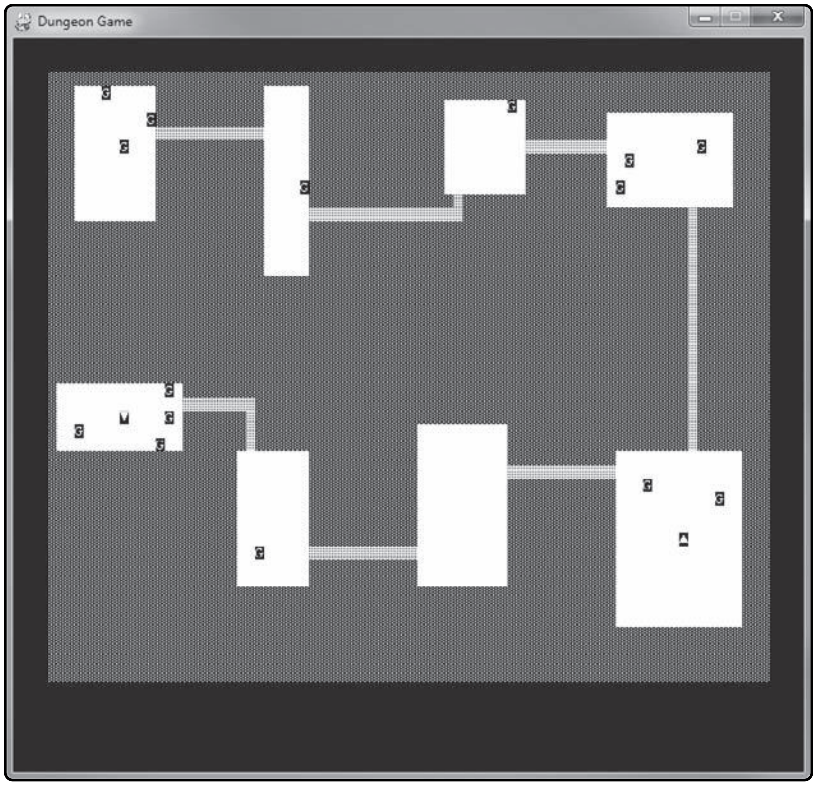


FIGURE 14.22

Random gold has been added to the level.

```
drops = random.randint(5,20)
for n in range(1,drops):
    tile = 175
    while tile == 175:
        x = random.randint(0,79)
        y = random.randint(0,44)
        tile = self.getCharAt(x,y)
    self.setCharAt(x,y,70) #'G'
```

HINT

Don't be concerned with dropping items on top of each other with this random code. It won't happen! The algorithms look exclusively for a *room code* for the item's position, so a previously dropped item at any location will have changed the code already.

Adding Weapons, Armor, and Health Potions

In addition to gold, we want to give the player a few random item drops here and there to make the dungeon seem to have been explored before, to give it some character. Usually there will be one or two weapon and armor items here and there in each level, perhaps where a poor adventurer met his fate long ago. In some games, monsters will drop items when you kill them. Using the technique shown above to add gold to the level, you can add any item you wish to the level in a similar way. You could even booby-trap some items, so that the player takes damage after picking up an item! To pick up an item or gold, of course, just add it to the player's gold count or inventory, and remove it from the level by setting that tile to a room or hall code.

So, let's just add one "W" and one "A" to the level, representing one weapon and one armor item, and two "H" items representing health potions (for healing). You can use this code to add any other items you wish. There are even some ASCII codes that look sort of like these items (if you use your imagination!).We don't need to keep track of the strengths of these items, or their value, because those random numbers can just be generated when the player picks them up! See the section titled "Advanced Gameplay" for more details on picking up items.

The code to drop an item is the same for gold, so we have some code again that needs to be put into a reusable method. Let's take the "gold code" and write a new method called `Dungeon.putCharInRandomRoom()`. This code can be reused endlessly to add anything you want in the dungeon, including monsters. In the lower levels, you might want to limit the health potions, or make them only restore a small amount of health, to increase the difficulty of the game. After all, the player gets essentially free gear upgrades by just finding those items, so don't make it *too easy*!

```
    self.putCharInRandomRoom(roomChar,86) #'W'
    self.putCharInRandomRoom(roomChar,64) #'A'
    self.putCharInRandomRoom(roomChar,71) #'H'
    self.putCharInRandomRoom(roomChar,71) #'H'
```

Here is the new method:

```
def putCharInRandomRoom(self,targetChar,itemChar):
    tile = 0
    while tile != targetChar:
        x = random.randint(0,79)
        y = random.randint(0,44)
        tile = self.getCharAt(x,y)
    self.setCharAt(x,y,itemChar)
```

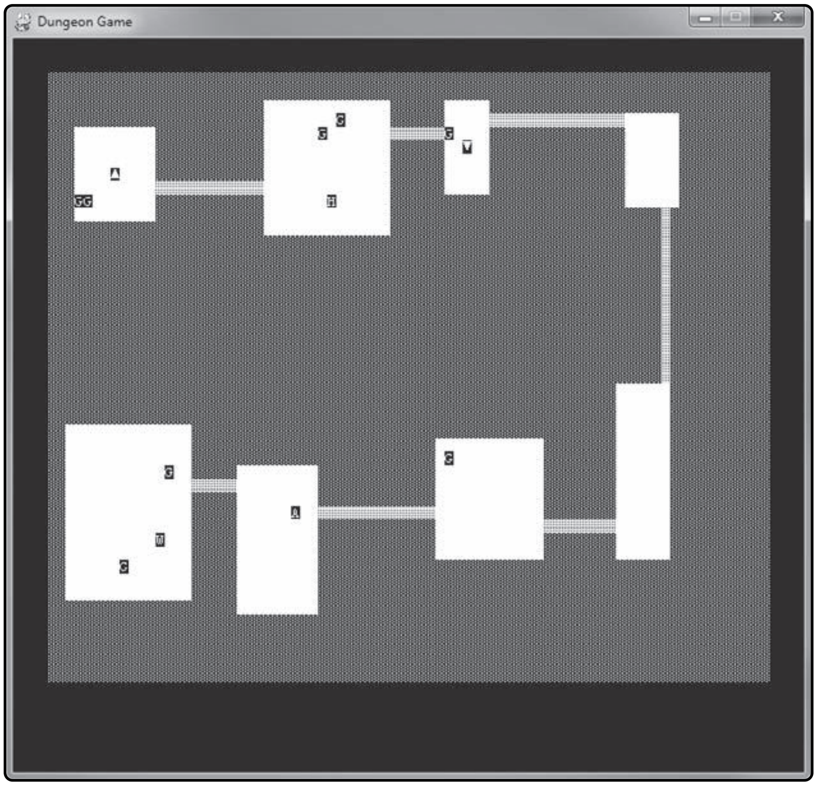


FIGURE 14.23
Can you spot the [W]eapon, [A]rmor, and [H]ealth potion items?

Adding Monsters

Ah, monsters! We need a good, strong antagonist to make a story interesting. Monsters can be represented with any ASCII code, so peruse the table and find some really gnarly looking characters for your monsters! Then add them using the same code we wrote for the gold. Monsters should look scary. Choose the scariest-looking ASCII codes you can for the monsters. For instance, any of these characters might be a classic one-eyed "beholder": ◘, ○, ◙, @. A "basilisk" might be represented with: ß or B. In the classic *Roguelike* gameplay, you want the monsters to get stronger and scarier as the player reaches deeper levels, and start with basic scary animals near the top, like giant rats, rabid wolves, etc.

Due to the additional work required, we might want to manage the monsters in the main program rather than inside the `Dungeon` class. But just to get things started, here's some code that will add several "M" characters to the dungeon. Remember, the character is just a placeholder, without any data behind it. When the player encounters the monster, all of its info will be generated on the fly.

```
    num = random.randint(5,10)
    for n in range(0,num):
        self.putCharInRandomRoom(roomChar,20)
```

Complete Dungeon Class

That's a lot of code we've gone over in a short time. The complete source code for this game can be found in the Chapter 14 resource files (found at www.courseptr.com/downloads). We'll go over the final code for the game later in this chapter. Right now, let's just see a complete listing of the `Dungeon` class now that it's finished. The file is called Dungeon.py.

```
import sys, time, random, math, pygame
from pygame.locals import *
from MyLibrary import *

class Dungeon():
    def __init__(self,offsetx,offsety):
        #create the font sprite
        self.text = MySprite()
        self.text.load("ascii8x12.png", 8, 12, 32)
        #create the level list
        self.tiles = list()
        for n in range(0,80*45):
            self.tiles.append(-1)
        self.offsetx = offsetx
        self.offsety = offsety
        self.generate()

    def generate(self, emptyChar=175, roomChar=218, hallChar=177):
        self.emptyChar = emptyChar
        self.roomChar = roomChar
        self.hallChar = hallChar

        #clear existing level
        for index in range(0,80*45):
            self.tiles[index] = emptyChar

        #create random rooms
        PL = 4
```

```
PH = 8
SL = 5
SH = 14
self.rooms = list()
self.createRoom(0,0,PL,PH,SL,SH)
self.createRoom(20,0,PL,PH,SL,SH)
self.createRoom(40,0,PL,PH,SL,SH)
self.createRoom(60,0,PL,PH,SL,SH)
self.createRoom(0,22,PL,PH,SL,SH)
self.createRoom(20,22,PL,PH,SL,SH)
self.createRoom(40,22,PL,PH,SL,SH)
self.createRoom(60,22,PL,PH,SL,SH)

#connect the rooms with halls
self.createHallRight(self.rooms[0],self.rooms[1],hallChar)
self.createHallRight(self.rooms[1],self.rooms[2],hallChar)
self.createHallRight(self.rooms[2],self.rooms[3],hallChar)
self.createHallRight(self.rooms[4],self.rooms[5],hallChar)
self.createHallRight(self.rooms[5],self.rooms[6],hallChar)
self.createHallRight(self.rooms[6],self.rooms[7],hallChar)

#choose a random northern room to connect with the south
choice = random.randint(0,3)
print("choice:" + str(choice) + "," + str(choice+4))
self.createHallDown(self.rooms[choice],self.rooms[choice+4],hallChar)

#add rooms to level
for room in self.rooms:
    for y in range(room.y,room.y+room.height):
        for x in range(room.x,room.x+room.width):
            self.setCharAt(x, y, roomChar)

#add entrance portal
choice = random.randint(0,7)
self.entrance_x = self.rooms[choice].x + self.rooms[choice].width//2
self.entrance_y = self.rooms[choice].y + self.rooms[choice].height//2
self.setCharAt(self.entrance_x,self.entrance_y,29) #entry portal
print("entrance:",choice,self.entrance_x,self.entrance_y)
```

```
    #add entrance and exit portals
    choice2 = random.randint(0,7)
    while choice2 == choice:
        choice2 = random.randint(0,7)
    x = self.rooms[choice2].x + self.rooms[choice2].width//2
    y = self.rooms[choice2].y + self.rooms[choice2].height//2
    self.setCharAt(x,y,30) #exit portal
    print("exit:",choice2,x,y)

    #add random gold
    drops = random.randint(5,20)
    for n in range(1,drops):
        self.putCharInRandomRoom(roomChar,70) #'G'

    #add weapon, armor, and health potiions
    self.putCharInRandomRoom(roomChar,86) #'W'
    self.putCharInRandomRoom(roomChar,64) #'A'
    self.putCharInRandomRoom(roomChar,71) #'H'
    self.putCharInRandomRoom(roomChar,71) #'H'

   #add some monsters
    num = random.randint(5,10)
    for n in range(0,num):
        self.putCharInRandomRoom(roomChar,20)

def putCharInRandomRoom(self,targetChar,itemChar):
    tile = 0
    while tile != targetChar:
        x = random.randint(0,79)
        y = random.randint(0,44)
        tile = self.getCharAt(x,y)
    self.setCharAt(x,y,itemChar)

def createRoom(self,x,y,rposx,rposy,rsizel,rsizeh):
    room = Rect(x + random.randint(1,rposx),
                y + random.randint(1,rposy),
                random.randint(rsizel,rsizeh),
                random.randint(rsizel,rsizeh))
    self.rooms.append(room)

def createHallRight(self,src,dst,hallChar):
    pathx = src.x + src.width
    pathy = src.y + random.randint(1,src.height-2)
    self.setCharAt(pathx,pathy,hallChar)
    if pathy > dst.y and pathy < dst.y + dst.height:
        while pathx < dst.x:
            pathx += 1
            self.setCharAt(pathx,pathy,hallChar)
    else:
        while pathx < dst.x+1:
            pathx += 1
            self.setCharAt(pathx,pathy,hallChar)
        if pathy < dst.y+1:
            self.setCharAt(pathx,pathy,hallChar)
            while pathy < dst.y:
                pathy += 1
                self.setCharAt(pathx,pathy,hallChar)
        else:
            self.setCharAt(pathx,pathy,hallChar)
            while pathy > dst.y + dst.height:
                pathy -= 1
                self.setCharAt(pathx,pathy,hallChar)

def createHallDown(self,src,dst,hallChar):
    pathx = src.x + random.randint(1,src.width-2)
    pathy = src.y + src.height
    self.setCharAt(pathx,pathy,hallChar)
    if pathx > dst.x and pathx < dst.x + dst.width:
        while pathy < dst.y:
            pathy += 1
            self.setCharAt(pathx,pathy,hallChar)
    else:
        while pathy < dst.y+1:
            pathy += 1
            self.setCharAt(pathx,pathy,hallChar)
        if pathx < dst.x+1:
```

```
            self.setCharAt(pathx,pathy,hallChar)
            while pathx < dst.x:
                pathx += 1
                self.setCharAt(pathx,pathy,hallChar)
        else:
            self.setCharAt(pathx,pathy,hallChar)
            while pathx > dst.x + dst.width:
                pathx -= 1
                self.setCharAt(pathx,pathy,hallChar)

def getCharAt(self, x, y):
    if x < 0 or x > 79 or y < 0 or y > 44:
        print("error: x,y = ", x, y)
        return
    index = y * 80 + x
    if index < 0 or index > 80*45:
        print("error: index = ", index)
        return
    return self.tiles[index]

def setCharAt(self, x, y, char):
    if x < 0 or x > 79 or y < 0 or y > 44:
        print("error: x,y = ", x, y)
        return
    index = y * 80 + x
    if index < 0 or index > 80*45:
        print("error: index = ", index)
        return
    self.tiles[index] = char

def draw(self, surface):
    for y in range(0,45):
        for x in range(0,80):
            char = self.getCharAt(x,y)
            if char >= 0 and char <= 255:
                self.draw_char(surface, x, y, char)
            else:
                pass #empty tile
def draw_char(self, surface, tilex, tiley, char):
    self.text.X = self.offsetx + tilex * 8
    self.text.Y = self.offsety + tiley * 12
    self.text.frame = char
    self.text.last_frame = char
    self.text.update(0)
    self.text.draw(surface)
```

Adding the Player's Character

We now have a playable random dungeon level generator, and have populated the dungeon with things. The player's character (PC) is a special character that will not be merely added to the dungeon; it will be maintained with separate variables. After all, we have to keep track of the player's stats. This is best done with a custom `Player` class. When the game begins, usually the player can participate in "rolling" the character's stats. We will just fill them in randomly for the chapter example, but you may want to add a feature to your own game where you can customize the player or re-roll the stats before moving on.

Our player variable will be global, not part of the `Dungeon` class. So, let's assume the player object is created before the dungeon object (in the main program code), and go from there. When the level is generated, we can just position the player using the `Dungeon.entrance_x` and `Dungeon.entrance_y` variables. We'll skimp on some of the code at this point because the complete source code for the main file is shown later.

```
    dungeon.generate()
    player.x = dungeon.entrance_x+1
    player.y = dungeon.entrance_y+1
```

The Player Class

The `Player` class has all of the variables and methods needed to manage the player's stats and position. We will draw the player's ASCII character separately from the rest of the dungeon, over the top of the dungeon tiles. Here, now, is a highly developed `Player` class with a helper function called `Die` used to generate random numbers like from a die roll. The class will be added to a new Python source code file called Player.py. Just to be thorough, note the required `import` statements. Note that the `Monster` class has been appended to the bottom of this listing, as it inherits directly from `Player`.

```
import sys, time, random, math, pygame
from pygame.locals import *
from MyLibrary import *
from Dungeon import *
```

```
def Die(faces):
    roll = random.randint(1,faces)
    return roll

class Player():
    def __init__(self,dungeon,level,name):
        self.dungeon = dungeon
        self.alive = True
        self.x = 0
        self.y = 0
        self.name = name
        self.gold = 0
        self.experience = 0
        self.level = level
        self.weapon = level
        self.weapon_name = "Club"
        self.armor = level
        self.armor_name = "Rags"
        self.roll()

    def roll(self):
        self.str = 6 + Die(6) + Die(6)
        self.dex = 6 + Die(6) + Die(6)
        self.con = 6 + Die(6) + Die(6)
        self.int = 6 + Die(6) + Die(6)
        self.cha = 6 + Die(6) + Die(6)
        self.max_health = 10 + Die(self.con)
        self.health = self.max_health

    def levelUp(self):
        self.str += Die(6)
        self.dex += Die(6)
        self.con += Die(6)
        self.int += Die(6)
        self.cha += Die(6)
        self.max_health += Die(6)
        self.health = self.max_health
```

```
def draw(self,surface,char):
    self.dungeon.draw_char(surface,self.x,self.y,char)

def move(self,movex,movey):
    char = self.dungeon.getCharAt(self.x + movex, self.y + movey)
    if char not in (self.dungeon.roomChar,self.dungeon.hallChar):
        return False
    else:
        self.x += movex
        self.y += movey
        return True

def moveUp(self): return self.move(0,-1)
def moveDown(self): return self.move(0,1)
def moveLeft(self): return self.move(-1,0)
def moveRight(self): return self.move(1,0)

def addHealth(self,amount):
    self.health += amount
    if self.health < 0:
        self.health = 0
    elif self.health > self.max_health:
        self.health = self.max_health

def addExperience(self,xp):
    cap = math.pow(10,self.level)
    self.experience += xp
    if self.experience > cap:
        self.levelUp()

def getAttack(self):
    attack = self.str + Die(20)
    return attack

def getDefense(self):
    defense = self.dex + self.armor
    return defense
    def getDamage(self,defense):
        damage = Die(8) + self.str + self.weapon - defense
        return damage

class Monster(Player):
    def __init__(self,dungeon,level,name):
        Player.__init__(self,dungeon,level,name)
        self.gold = random.randint(1,4) * level
        self.str = 1 + Die(6) + Die(6)
        self.dex = 1 + Die(6) + Die(6)
```

Moving the Player Character

Now we have enough built up to move and draw the player's *character* (which brings a whole new meaning to the word!). In the event handler of the main program, we'll respond to key press events in order to move the player's character. What happens is, we try to move the player in one of the four directions. If that way is blocked by any code other than a room or hall code, then that is an object and we should respond to it before moving over it. Note that some of the code for the event handler has been skipped so we can focus on the important stuff.

```
    if event.key == K_ESCAPE: sys.exit()

    elif event.key == K_SPACE:
        dungeon.generate(TILE_EMPTY,TILE_ROOM,TILE_HALL)
        player.x = dungeon.entrance_x+1
        player.y = dungeon.entrance_y+1

    elif event.key==K_UP or event.key==K_w:
        if player.moveUp() == False:
            playerCollision(0,-1)

    elif event.key==K_DOWN or event.key==K_s:
        if player.moveDown() == False:
            playerCollision(0,1)

    elif event.key==K_RIGHT or event.key==K_d:
        if player.moveRight() == False:
            playerCollision(1,0)
    elif event.key==K_LEFT or event.key==K_a:
        if player.moveLeft() == False:
            playerCollision(-1,0)
```

When the up, down, left, or right keys are pressed, we "simulate" movement in that direction by calling `Player.MoveUp()`, `Player.MoveDown()`, and so on. These methods return `True` if the movement is legal, or `False` if there's an obstacle in the way. If that happens, we need to respond to the "collision" with the obstacle. That is handled by a helper function in our main program, called `playerCollision()`. This is some early code that just tries to identify the obstacle and print a message to the console. But, this is all we need to complete the game. Now, if the player tries to move into a wall, the game won't let them!

```
def playerCollision(stepx,stepy):
    global TILE_EMPTY,TILE_ROOM,TILE_HALL,dungeon,player,level
    char = dungeon.getCharAt(player.x + stepx, player.y + stepy)
    if char == 29: #portal up
        print("portal up")
    elif char == 30: #portal down
        print("portal down")
    elif char == TILE_EMPTY: #wall
        print("You ran into the wall--ouch!")
```

Advanced Gameplay

In this section, we cover the advanced gameplay options that bring the game to life, covering fighting, visibility, item pickups, A.I. movement, and others. By this point, we have a fully interactive game but incomplete gameplay with the finished `Dungeon`, `Player`, and `Monster` classes. To remedy the gameplay issue, we'll include a complete listing of the code, one section at a time. Let's begin with the initialization. In order for this program to run, be sure to include the Dungeon.py and Player.py files in the same folder, along with the ASCII font file, ascii8x12.png.

```
import sys, time, random, math, pygame
from pygame.locals import *
from MyLibrary import *
from Dungeon import *
from Player import *

def game_init():
    global screen, backbuffer, font1, font2, timer
    pygame.init()
    screen = pygame.display.set_mode((700,650))
    backbuffer = pygame.Surface((700,650))
    pygame.display.set_caption("Dungeon Game")
    font1 = pygame.font.SysFont("Courier New", size=18, bold=True)
    font2 = pygame.font.SysFont("Courier New", size=14, bold=True)
    timer = pygame.time.Clock()

def Die(faces):
    roll = random.randint(1,faces)
    return roll
```

Picking Up Items

To enable item pickups, the player must have an inventory, which presumes he or she has a backpack of some sort. We can make that assumption if we want, but an inventory system requires quite a bit of design and forethought. We can't just add items to a list. How do we display them? There's no feature in the game to display an inventory system. Perhaps a secondary screen that hides the dungeon and shows the inventory? That's a feasible idea, but not one we're going to explore in this chapter. I will encourage you to pursue this idea if you wish, but we'll stick with the simple approach with weapons and armor.

The simple approach is this: The player has an attack and defense value for fighting monsters. When you pick up a weapon or armor in the dungeon, if that item is better than what you currently have, then it is auto-equipped. If not, it is turned into gold. A more complex *Rogue-like* game would make you return to a shop and sell the items, but that is another rather complex feature that would take a very long time to explain, and so it is beyond the scope of this chapter. We need a whole book to explore these ideas! In fact, whole books *have* been written on them.

Previously, we added several items to the game ("W," "A," "G," and "H"), so let's deal with those first, and then you can use similar code for any other types of items you wish to add to the game. To do this, we return to the `playerCollision()` function again, in the main program code. Here's an example of picking up gold.

```
def playerCollision(stepx,stepy):
    global TILE_EMPTY,TILE_ROOM,TILE_HALL,dungeon,player,level
    yellow = (220,220,0)
    green = (0,220,0)

    #get object at location
    char = dungeon.getCharAt(player.x + stepx, player.y + stepy)
    if char == 29: #portal up
         message("portal up")

     elif char == 30: #portal down
         message("portal down")

     elif char == TILE_EMPTY: #wall
         message("You ran into the wall--ouch!")

     elif char == 70: #gold
         gold = random.randint(1,level)
         player.gold += gold
         dungeon.setCharAt(player.x+stepx, player.y+stepy, TILE_ROOM)
         message("You found " + str(gold) + " gold!", yellow)
```

To handle weapons, we need to look for the ASCII code used for a weapon drop. In the `Dungeon` class, this was the "W" character, ASCII code 86. We'll write some code to give the player a random new weapon when they pick up a "W."

```
    elif char == 86: #weapon
        weapon = random.randint(1,level+2)
        if level <= 5: #low levels get crappy stuff
            temp = random.randint(0,2)
        else:
            temp = random.randint(3,6)
        if temp == 0: name = "Dagger"
        elif temp == 1: name = "Short Sword"
        elif temp == 2: name = "Wooden Club"
        elif temp == 3: name = "Long Sword"
        elif temp == 4: name = "War Hammer"
        elif temp == 5: name = "Battle Axe"
        elif temp == 6: name = "Halberd"
        if weapon >= player.weapon:
            player.weapon = weapon
            player.weapon_name = name
            message("You found a " + name + " +" + str(weapon) + "!",yellow)
        else:
            player.gold += 1
            message("You discarded a worthless " + name + ".")
        dungeon.setCharAt(player.x+stepx, player.y+stepy, TILE_ROOM)
```

We'll use similar code for armor pickups. The code for an armor item is "A," 64.

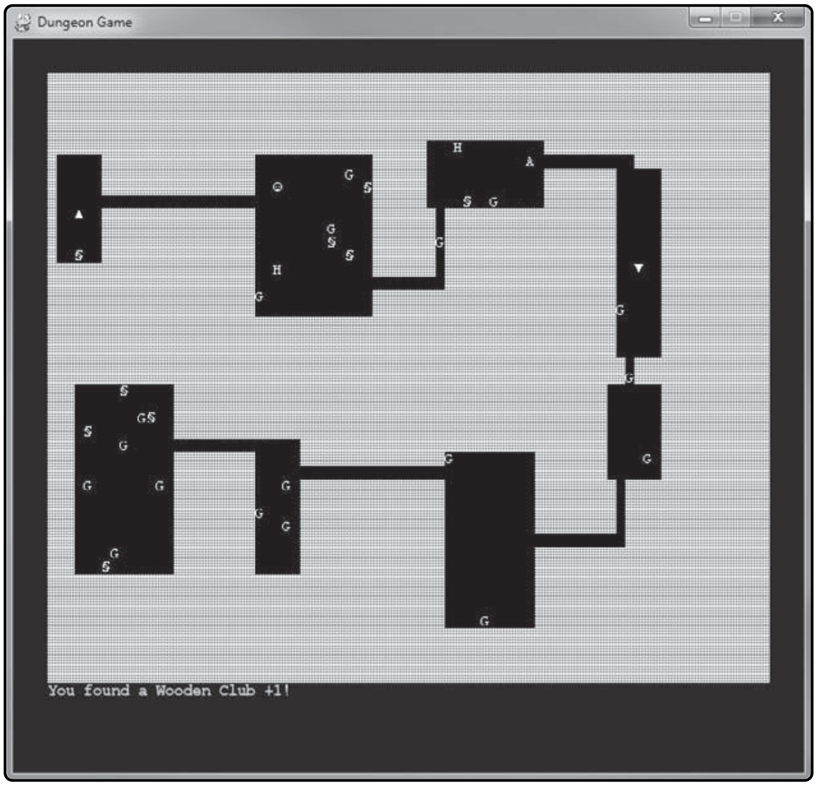
```
elif char == 64: #armor
    armor = random.randint(1,level+2)
    if level <= 5: #low levels get crappy stuff
        temp = random.randint(0,2)
    else:
        temp = random.randint(3,7)
    if temp == 0: name = "Cloth"
    elif temp == 1: name = "Patchwork"
    elif temp == 2: name = "Leather"
    elif temp == 3: name = "Chain"
    elif temp == 4: name = "Scale"
    elif temp == 5: name = "Plate"
    elif temp == 6: name = "Mithril"
    elif temp == 7: name = "Adamantium"
    if armor >= player.armor:
        player.armor = armor
        player.armor_name = name
        message("You found a " + name + " +" + str(armor) + "!",yellow)
    else:
        player.gold += 1
        message("You discarded a worthless " + name + ".")
    dungeon.setCharAt(player.x+stepx, player.y+stepy, TILE_ROOM)
```

Lastly, we have health potions to pick up, character "H," 71.

```
elif char == 71: #health
    heal = 0
    for n in range(0,level):
        heal += Die(6)
    player.addHealth(heal)
    dungeon.setCharAt(player.x+stepx, player.y+stepy, TILE_ROOM)
    message("You drank a healing potion worth " + str(heal) + \
        " points!", green)
```

It's been a while since we've seen a build of the game, so let's see what it looks like at this point. The `message()` function is a helper that displays action messages below the dungeon. Just for the variety, the room character has been replaced with a blank space, which is black.

Now the rooms are black and the outer dungeon is solid-looking. You may use either theme or use another one entirely if you wish. This just shows the variety of options.



```
You found a Wooden Club +1!
```

FIGURE 14.24
Picking up a new weapon.

Fighting Monsters

Combat in most RPGs follows very specific rules that we will try to emulate. The basic premise for combat in modern RPGs may differ from the way it was done in *Rogue*, but we'll do our best to make combat fun. There are three factors involved in combat:

- Defender's defense value
- Attacker's attack value
- Attacker's damage value

You might have noticed that we already have methods in the `Player` class for doing these things, so we're all ready to go as far as the code is concerned. Let's dig into the calculations to understand how combat works.

Calculating Defense

To calculate the defense value used for a to-hit roll, which determines whether the attack succeeds, we use the following formula:

```
Defense = DEXTERITY + Armor Value
```

Let's face it, we're going to fake it here regarding monster armor, and just throw a random number out for use as armor, based on the current dungeon level. So, given the current dungeon level, we'll multiply that by a random number. Let's say the current level is 5. Therefore, whatever number we roll for the monster's dexterity will be multiplied by 5.

Calculating Attack

The formula for the attack value is as follows:

```
Attack = STRENGTH + D20
```

The "D20" means rolling a 20-sided die. There's a wide variance in the attack value with such a many-sided die. This reflects the effect of combat, where some swings totally miss and some hit the enemy. Based on this attack value, we compare it to the monster's defense value. If attack is greater than defense, then the "to-hit" roll succeeds. Damage is calculated next.

Calculating Damage

If the to-hit roll succeeds, then the defender takes damage. That amount will be based on the following calculation:

```
Damage = D8 + STRENGTH + Weapon Damage - Defense
```

As you can see, the `Defense` value is used twice: first, to calculate to-hit, and secondly, to calculate damage. That's good! It means the stats of the defender are very important to the gameplay.

Combat Rounds

The game will only recognize one fight at a time. When the player attacks a monster, there's no turning back, no way to flee. A global *monster* object will remember the current monster being fought.

```
monster = Monster(dungeon,level,"Grue")
```

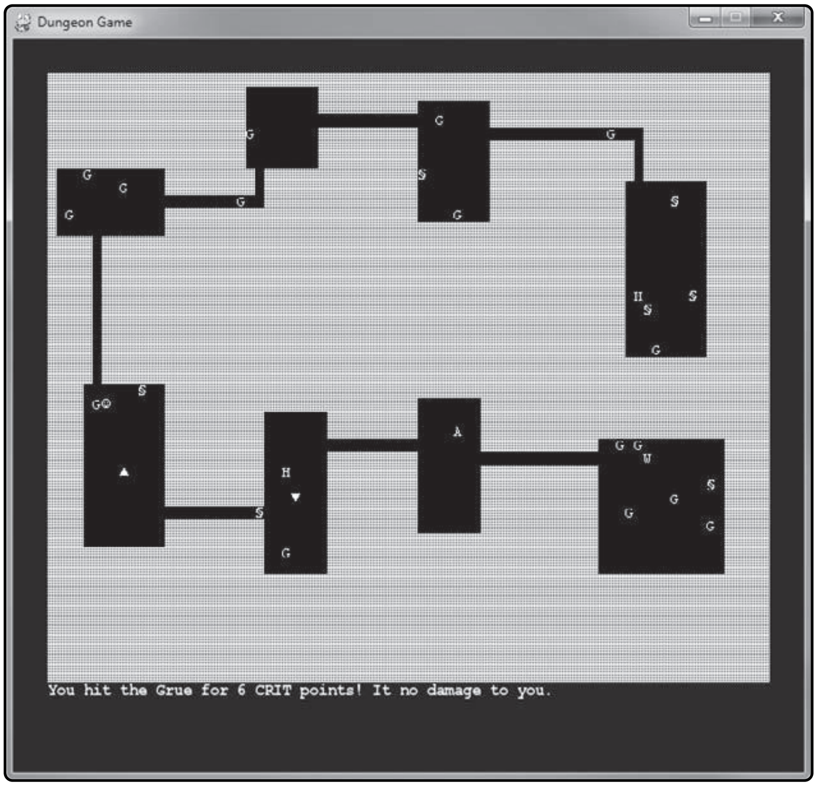
Hitting a Monster

Unfortunately, to build a really comprehensive combat system like this requires that we keep track of each monster individually in a list, which is just too overwhelming for this single chapter project. So, we're going to cheat a bit. All of the logic for rolling to-hit chance, damage, and so on, could be written given the methods are already written in `Player`, but one hit will kill every monster. Where we'll make it difficult is in the amount of damage that the player takes as a result. Some high-level monsters could potentially kill the player with a single blow too, if the player's armor class is not up to the pounding. The only time an attack will continue for more than one round is if the player *misses completely*!

We can use the `playerCollision()` function again to attack a monster.

```
elif char == 20: #monster
    attack_monster(player.x+stepx, player.y+stepy, 20)
```

Let's write that important function now. Working with the to-hit, attack, damage, defense values is a lot of fun! The combat calculations are often the most enjoyable part of building an RPG. If you want to make improvements or change the logic, you are welcome to make this game *your own*. Figure 14.25 shows the result of a battle with a Grue! Note that gold was dropped where the Grue died. Can you spot the player? It's in the room on the lower left corner.



```
You hit the Grue for 6 CRIT points! It no damage to you.
```

FIGURE 14.25

Fighting a nasty Grue in the dungeon.

```
def attack_monster(x,y,char):
    global dungeon, TILE_ROOM
    monster = Monster(dungeon,level,"Grue")

    #player's attack
    defense = monster.getDefense()
    attack = player.getAttack()
    damage = player.getDamage(defense)
    battle_text = "You hit the " + monster.name + " for "
    if attack == 20 + player.str: #critical hit?
        damage *= 2
        battle_text += str(damage) + " CRIT points!"
        dungeon.setCharAt(x, y, 70) #drop gold
    elif attack > defense: #to-hit?
        if damage > 0:
            battle_text += str(damage) + " points."
            dungeon.setCharAt(x, y, 70) #drop gold
        else:
            battle_text += "no damage!"
            damage = 0
    else:
        battle_text = "You missed the " + monster.name + "!"
        damage = 0

    #monster's attack
    defense = player.getDefense()
    attack = monster.getAttack()
    damage = monster.getDamage(defense)
    if attack > defense: #to-hit?
        if damage > 0:
            #if damage is overwhelming, halve it
            if damage > player.max_health: damage /= 2
            battle_text += " It hit you for " + str(damage) + " points."
            player.addHealth(-damage)
        else:
            battle_text += " It no damage to you."
    else:
        battle_text += " It missed you."
```

```
#display battle results
message(battle_text)

#did the player survive?
if player.health <= 0: player.alive = False
```

Moving Monsters

When moving monsters, we have a lot of options, and it depends on what you want the monsters to do in the game. We really don't want them to just immediately start chasing the player. But what if the player stumbles upon a monster? If the player is close enough, the monster definitely should move toward the player! Just remember, this is a turn-based game, so things only happen when the player moves. We can borrow some of the visibility code to see when the player is close enough to a monster to trigger the A.I. code for movement. It's important to make sure the monster doesn't go through any walls or gold or items. If a monster moves over gold or other items in our "easy" dungeon tile algorithm, those items are erased by the monster's code.

Let's begin by inserting some logic into the user input section of code so that monsters will move when the player moves. This code is found in the main loop in the event handler.

```
elif event.key==K_UP or event.key==K_w:
    if player.moveUp() == False:
        playerCollision(0,-1)
    else:
        move_monsters()

elif event.key==K_DOWN or event.key==K_s:
    if player.moveDown() == False:
        playerCollision(0,1)
    else:
        move_monsters()

elif event.key==K_RIGHT or event.key==K_d:
    if player.moveRight() == False:
        playerCollision(1,0)
    else:
        move_monsters()
elif event.key==K_LEFT or event.key==K_a:
    if player.moveLeft() == False:
        playerCollision(-1,0)
    else:
        move_monsters()
```

Although the `Monster` class inherits from `Player`, and thus it has the move methods available, we can't use them! Monsters in the tilemap are just placeholders until the player stumbles into them. No, we have to move things in the dungeon with new code. With these two helper functions, `move_monsters()` and `move_monster()`, every time the player makes a move, the monsters all move too! Better yet, they move inside their current room and do not walk over anything or bump into any of the walls.

```
def move_monsters():
    #find monsters
    for y in range(0,44):
        for x in range(0,79):
            tile = dungeon.getCharAt(x,y)
            if tile == 20: #monster?
                move_monster(x,y,20)


def move_monster(x,y,char):
    global TILE_ROOM
    movex = 0
    movey = 0
    dir = random.randint(1,4)
    if dir == 1: movey = -1
    elif dir == 2: movey = 1
    elif dir == 3: movex = -1
    elif dir == 4: movex = 1
    c = dungeon.getCharAt(x + movex, y + movey)
    if c == TILE_ROOM:
        dungeon.setCharAt(x, y, TILE_ROOM) #delete old position
        dungeon.setCharAt(x+movex, y+movey, char) #move to new position
```

Visibility Range

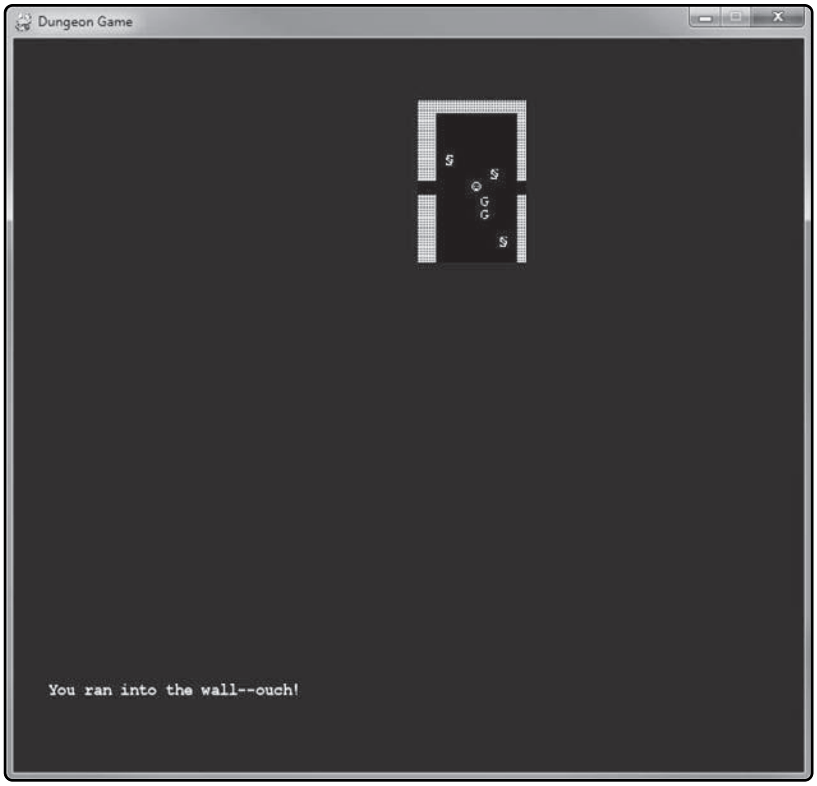
Let's just admit it: the game isn't really all that scary if you can see the whole level immediately from a bird's-eye view! What we need to implement is a scheme to hide everything that is beyond the player's visible range. Some *Roguelike* games tend to reveal the level and keep it visible as the player explores, as a sort of built-in mapping system. Some reveal the dungeon as you go, but darken areas that are no longer visible due to lighting. These are really great features that add to the depth of the gameplay. What can we do very simply to create a similar effect without getting bogged down in complex code? To hide everything not in view of the player requires a ray-casting system. It's not *hard* to write code to do that, but the code does require more explanation than we can get into in this single chapter.

TRICK

Learn how a line-of-sight ray-casting algorithm can be added to the game in my book *Visual C# Game Programming for Teens,* published in 2011 by Course PTR. That entire book is devoted to RPG techniques.

To reveal the level and keep it visible requires an extra flag attached to every tile that determines whether it has been "seen" or not. That would require modifications to the dungeon code, which I am not prepared to do at this point. We will take a simple but effective approach: simulating a torch light around the player. This adds an interesting new dimension to the gameplay. What if the player runs out of lamp oil or candles? We could reduce the player's visibility to a tiny circle, but enlarge it if the player has a light source. Wouldn't that be spooky, wandering around when you can only see a few feet in front of you? Only a fool would take the chance of running into a Grue in the dark!

This is surprisingly easier than one might expect it to be. All we have to do is go back to the `Dungeon.draw()` method, duplicate it, and skip any tiles that are a certain range from the player. Or, rather, pass the method new parameters, position and radius, and have it draw only the tiles within that range. Since this is a dramatic change to the game, we'll make it an option that doesn't have to be on all the time. We'll call this new method `Dungeon.draw_radius()`. This surprisingly small amount of code produces the dramatic result shown in Figure 14.26.



```
You ran into the wall--ouch!
```

FIGURE 14.26

Now you can only see as far as the player's lamp allows—ooh, scary!

```
def draw_radius(self, surface, rx, ry, radius):
    left = rx - radius
    if left < 0: left = 0
    top = ry - radius
    if top < 0: top = 0
    right = rx + radius
    if right > 79: right = 79
    bottom = ry + radius
    if bottom > 44: bottom = 44

    for y in range(top,bottom):
        for x in range(left,right):
            char = self.getCharAt(x,y)
            if char >= 0 and char <= 255:
                self.draw_char(surface, x, y, char)
```

Exiting the Level

The player may enter an entrance portal to go up a level, or an exit portal to go down a level. These "portals" might simply be stairs or steps between the levels, not really magical teleporters, but the game code treats them as such. Upon touching an entrance portal, the current level will be reduced by one, and the level re-generated. A real *Roguelike* game—that is, one intended for release with polish and detail—will keep track of the levels so the player doesn't get to cheat by just going in and out of a portal to re-fill the level with gold and items again. When a room has been cleared, it should remain cleared of items, but perhaps monsters could re-spawn after a while. We already detect when the player touches the entry or exit portals, so it's just a matter of updating the `level` variable and re-rolling the dungeon.

Wrapping Up the Gameplay

Now it's time to wrap up this game. We need to add a few basic things, like displaying the current level, the player's stats and gold, and other informative details. We will go over the basic code here, and you can use it to display any additional information you want in the game.

Printing Game Stats

The following function handles most of the information presented on the screen regarding the player's stats, current dungeon level, and so on.

```
def print_stats():
    print_text(font2, 0, 615, "STR")
    print_text(font2, 40, 615, "DEX")
    print_text(font2, 80, 615, "CON")
    print_text(font2, 120, 615, "INT")
    print_text(font2, 160, 615, "CHA")
    print_text(font2, 200, 615, "DEF")
    print_text(font2, 240, 615, "ATT")
    fmt = "{:3.0f}"
    print_text(font2, 0, 630, fmt.format(player.str))
    print_text(font2, 40, 630, fmt.format(player.dex))
    print_text(font2, 80, 630, fmt.format(player.con))
    print_text(font2, 120, 630, fmt.format(player.int))
    print_text(font2, 160, 630, fmt.format(player.cha))
    print_text(font2, 200, 630, fmt.format(player.getDefense()))
    #get average damage
    global att,attlow,atthigh
    att[0] = att[1]
    att[1] = att[2]
    att[2] = att[3]
    att[3] = att[4]
    att[4] = (player.getAttack() + att[0] + att[1] + att[2] + att[3]) // 5
    if att[4] < attlow: attlow = att[4]
    elif att[4] > atthigh: atthigh = att[4]
    print_text(font2, 240, 630, str(attlow) + "-" + str(atthigh))

    print_text(font2, 300, 615, "LVL")
    print_text(font2, 300, 630, fmt.format(player.level))
    print_text(font2, 360, 615, "EXP")
    print_text(font2, 360, 630, str(player.experience))

    print_text(font2, 440, 615, "WPN")
    print_text(font2, 440, 630, str(player.weapon) + ":" + player.weapon_name)
    print_text(font2, 560, 615, "ARM")
    print_text(font2, 560, 630, str(player.armor) + ":" + player.armor_name)

    print_text(font2, 580, 570, "GOLD " + str(player.gold))
    print_text(font2, 580, 585, "HLTH " + str(player.health) + "/" + \
        str(player.max_health))
```

Common Messages

The game needed a consistent way to display information about events happening in the game, like combat rolls and so forth. The `message()` function handles this, while the actual message is printed later.

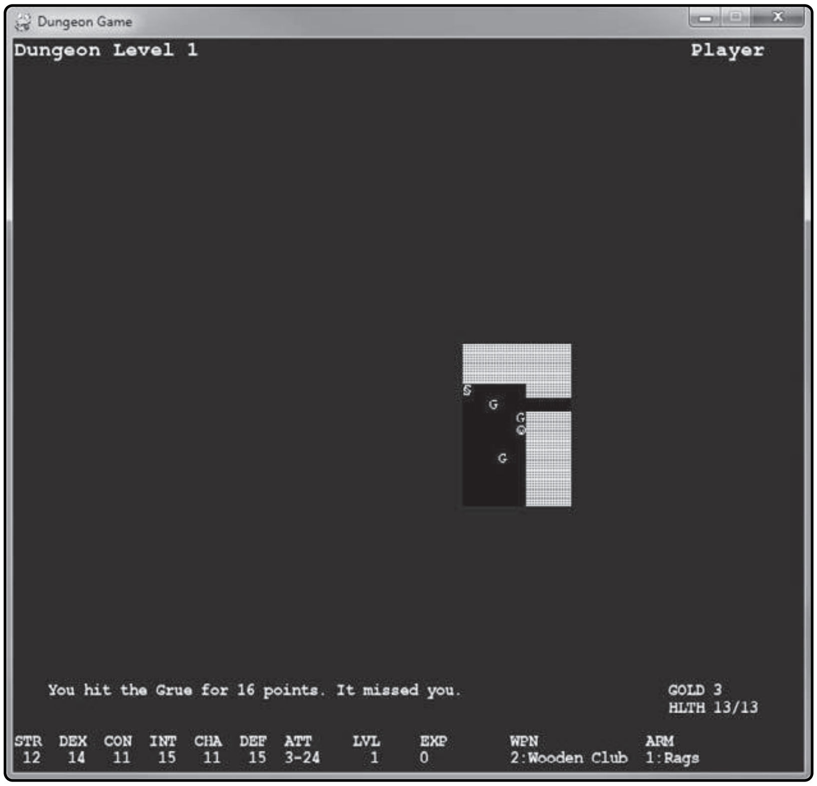
```
def message(text,color=(255,255,255)):
    global message_text, message_color
    message_text = text
    message_color = color
```

Remaining Code

Finally, we arrive at the main program logic in Game.py. Thus far, we have had two major helper classes in Player.py and Dungeon.py, which significantly cleaned up the code in Game.py, which we have been going over for the last few pages. Now, we can present just the core logic of the game. It is surprisingly simple considering how much is going on. You have already seen the event handler code, but it is shown here again so the code is not interrupted.

```
#define ASCII codes used for dungeon
TILE_EMPTY = 177
TILE_ROOM = 31
TILE_HALL = 31

#main program begins
game_init()
game_over = False
last_time = 0
dungeon = Dungeon(30, 30)
dungeon.generate(TILE_EMPTY,TILE_ROOM,TILE_HALL)
player = Player(dungeon, 1, "Player")
player.x = dungeon.entrance_x+1
player.y = dungeon.entrance_y+1
level = 1
message_text = "Welcome, brave adventurer!"
message_color = 0,200,50
draw_radius = False

#used to estimate attack damage
att = list(0 for n in range(0,5))
attlow=90
atthigh=0

#main loop
while True:
    timer.tick(30)
    ticks = pygame.time.get_ticks()

    #event section
    for event in pygame.event.get():
        if event.type == QUIT: sys.exit()
        elif event.type == KEYDOWN:
            if event.key == K_ESCAPE: sys.exit()
            elif event.key == K_TAB:
                #toggle map mode
                draw_radius = not draw_radius
            elif event.key == K_SPACE:
                dungeon.generate(TILE_EMPTY,TILE_ROOM,TILE_HALL)
                player.x = dungeon.entrance_x+1
                player.y = dungeon.entrance_y+1
            elif event.key==K_UP or event.key==K_w:
                if player.moveUp() == False:
                    playerCollision(0,-1)
                else:
                    move_monsters()
            elif event.key==K_DOWN or event.key==K_s:
                if player.moveDown() == False:
                    playerCollision(0,1)
                else:
                    move_monsters()
            elif event.key==K_RIGHT or event.key==K_d:
                if player.moveRight() == False:
                    playerCollision(1,0)
                else:
                    move_monsters()
            elif event.key==K_LEFT or event.key==K_a:
                if player.moveLeft() == False:
                    playerCollision(-1,0)
                else:
                    move_monsters()

    #clear the background
    backbuffer.fill((20,20,20))

    #draw the dungeon
    if draw_radius:
        dungeon.draw_radius(backbuffer, player.x, player.y, 6)
    else:
        dungeon.draw(backbuffer)

    #draw the player's little dude
    player.draw(backbuffer,0)
    #draw the back buffer
    screen.blit(backbuffer, (0,0))

    print_text(font1, 0, 0, "Dungeon Level " + str(level))
    print_text(font1, 600, 0, player.name)
    #special message text
    print_text(font2, 30, 570, message_text, message_color)
    print_stats()
    pygame.display.update()
```

With the lamp mode toggled on (toggle it with the Tab key) and all of the information displayed on the screen, our game is now finished! See Figure 14.27 for the final result.



```
Dungeon Level 1                                                   Player

   You hit the Grue for 16 points. It missed you.                GOLD 3
                                                                 HLTH 13/13
STR  DEX  CON  INT  CHA  DEF  ATT    LVL    EXP     WPN            ARM
 12   14   11   15   11   15  3-24    1     0       2:Wooden Club  1:Rags
```

FIGURE 14.27

Our RPG is finished with many of the features of a great *Roguelike* game.

Summary

This chapter was a singularly monumental effort to build an old-school RPG in a single go. I think we've succeeded in getting the most important aspects of the gameplay in, given the limited space. There's more that could be done with the game, of course, but that might be said of anything and everything when we're talking about game programming. I trust you will do something fun with it above and beyond the chapter example.

Challenges

1. **Make it possible to enter the portals and go up to the previous level, or down to the next level. To make this work, you will need to look into the random module to see how to set a common random number *seed* so that the random dungeon levels look the same each time the game is played. We don't want the levels to repeat forever, just during that single run.**
2. **Add some more monsters to the game, beyond the single group that has been added. Give them some more interesting movements by writing better A.I. code. For starters, monsters should chase and attack if the player gets too close!**
3. **Create a better user interface using bars for some of the stats (like health) instead of just numbers. Consider adding a limited inventory system and perhaps even a shop at the top level where the player can to go sell gear and get better stuff, heal damage, and so on.**

APPENDIX A

Installing Python and Pygame

This appendix shows step by step how to install Python and Pygame with figures detailing each step. Python and Pygame are both very easy to install, and equally easy to use, but someone who is not familiar with these tools may not know where to start. This appendix explains the steps.

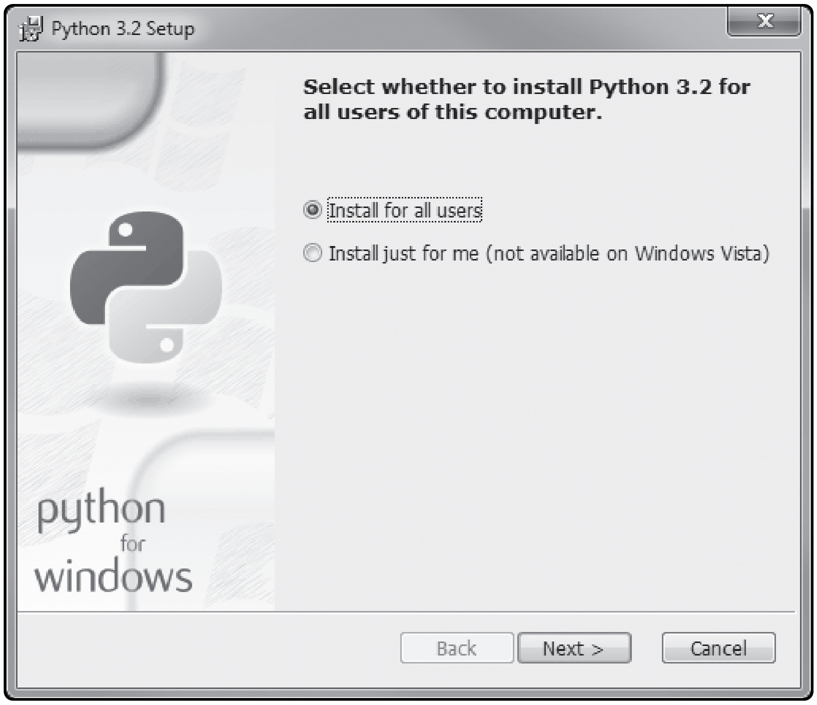
Installing Python

The website for Python is http://www.python.org. This book uses Python 3.2. If you have an earlier version already installed, like 2.7, the code in this book will *not* compile with that earlier 2.x version. The Python language changed with 3.0. If you have a later version after 3.2, then the code in this book *should* compile, but I can't guarantee it (obviously because that's in the future). Python is pretty easy to install, so I recommend installing 3.2 if you have the option of which version to install.

The download page for Python 3.2.1 is found at http://www.python.org/download/releases/3.2.1/.

If you are using Windows, you will want to download the "Windows x86" version. If you *know* for sure that you have 64-bit Windows, then download the "Windows x86-64" version. Likewise, if you have a Mac, download either the 32-bit or 64-bit version of the Python installer for Mac, depending on which version of OS X you have.

For easy reference, the following steps will go through the steps for installing Python (for beginners). Advanced readers may skip this section. First, when running the installer, we are presented with the first screen shown in Figure A-1.



Select whether to install Python 3.2 for all users of this computer.

[✓ Install for all users]

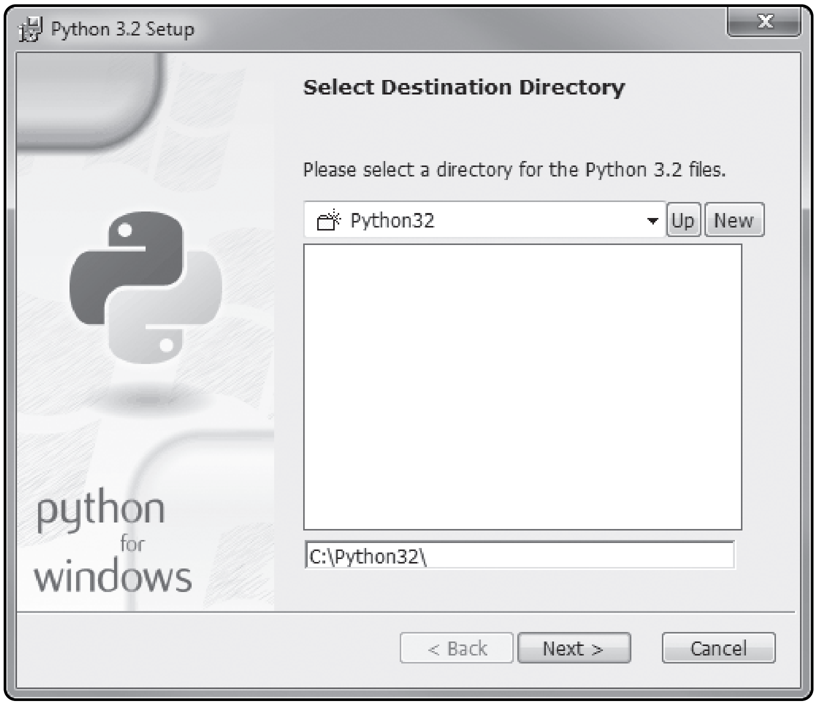
Install just for me (not available on Windows Vista)

FIGURE A.1

The first screen that comes up with the Python 3.2 installer.

The next screen, shown in Figure A.2, allows you to change the default installation folder. I recommend leaving it at the default location, for the benefit of file associations, but you may change it if you need to.

Next up is the installation options screen shown in Figure A.3. I recommend just leaving these options at their default values, unless you have a reason to change them.


Python 3.2 Setup

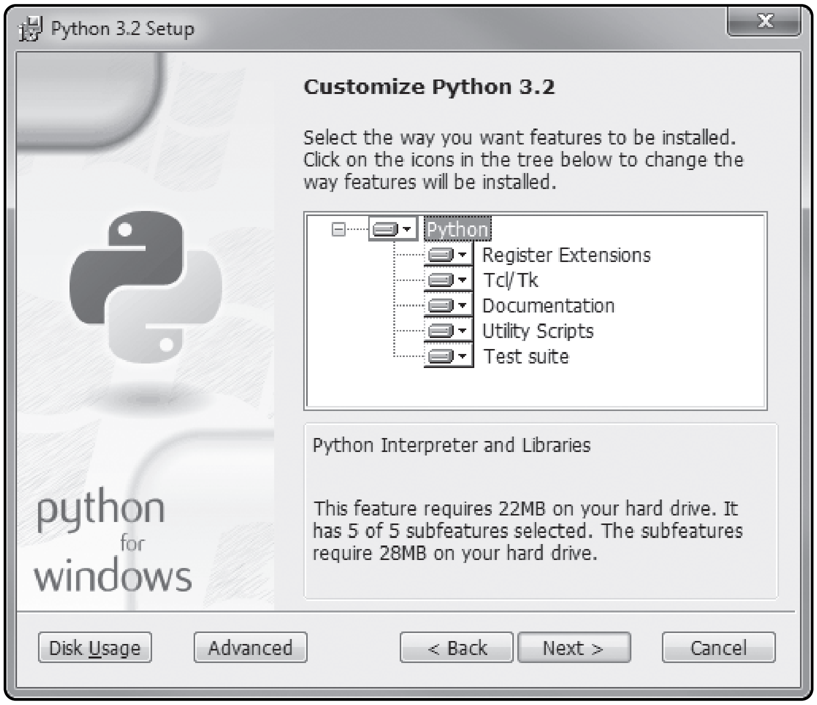
Select Destination Directory

Please select a directory for the Python 3.2 files.

Python32

C:\Python32\

Figure A.2 Selecting the installation folder.


Python 3.2 Setup

Customize Python 3.2

Select the way you want features to be installed.
Click on the icons in the tree below to change the
way features will be installed.

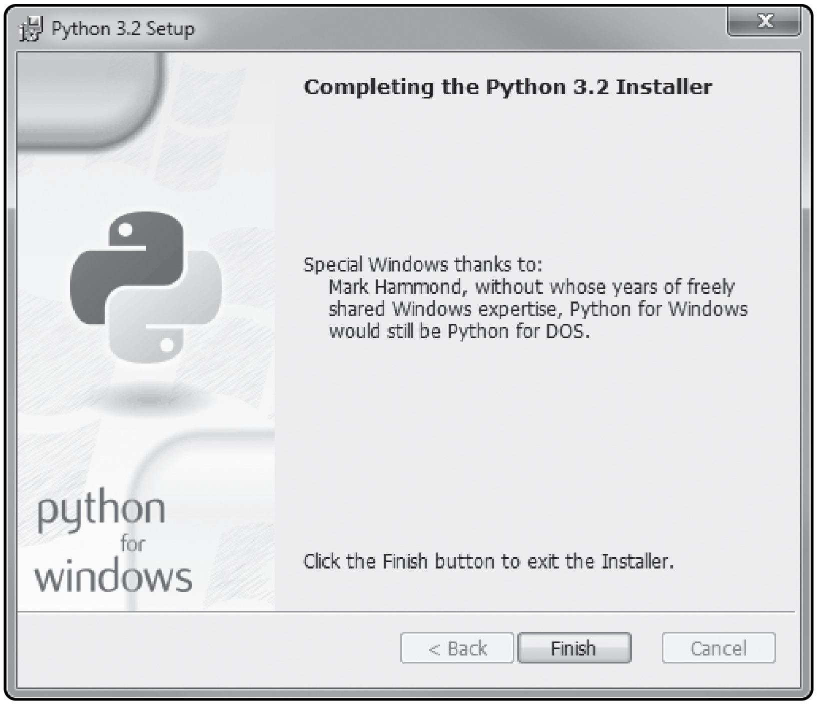
- Python
  - Register Extensions
  - Tcl/Tk
  - Documentation
  - Utility Scripts
  - Test suite

Python Interpreter and Libraries

This feature requires 22MB on your hard drive. It
has 5 of 5 subfeatures selected. The subfeatures
require 28MB on your hard drive.

Figure A.3 Installation options.

Next, we are presented with the final screen before installation begins, shown in Figure A.4. Upon clicking the Finish button, the installer will install the files into the location you specified.



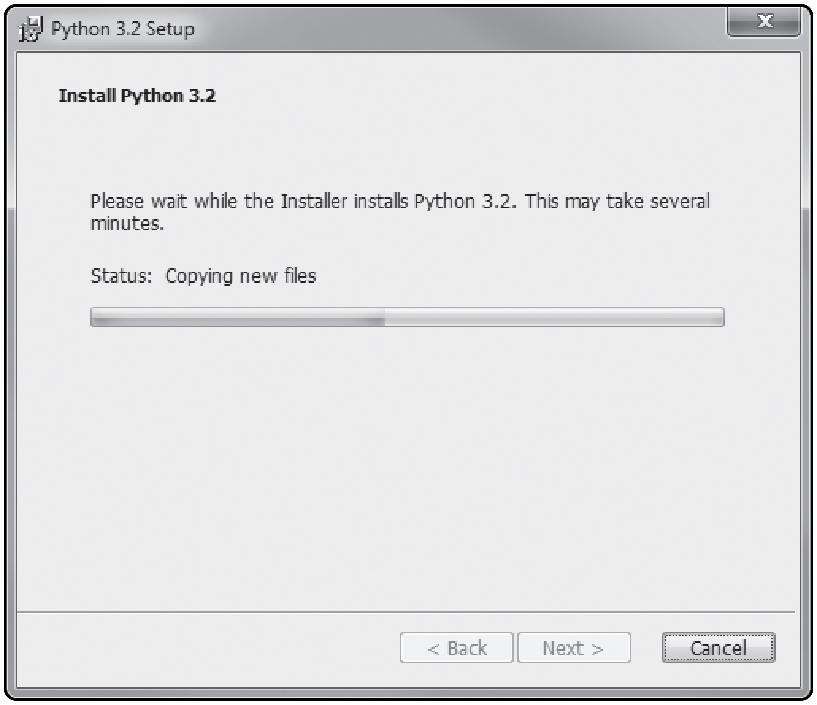
Completing the Python 3.2 Installer

Special Windows thanks to:
Mark Hammond, without whose years of freely shared Windows expertise, Python for Windows would still be Python for DOS.

Click the Finish button to exit the Installer.

FIGURE A.4
Preparing to begin the installation.

Finally, we see that the installation begins, as shown in Figure A.5. This will simply show the progress as the Python files are installed on your computer.


Python 3.2 Setup

Install Python 3.2

Please wait while the Installer installs Python 3.2. This may take several minutes.

Status: Copying new files

FIGURE A.5

Installation of the Python files.

INSTALLING PYGAME

Pygame must be installed *after* Python, because it is an add-on library. Pygame does not automatically get installed with Python. Despite the name, Pygame was not created by the same developers that created Python itself. So, we must download and install Pygame separately. This must be done after Python has already been installed.

There may be a new version of Pygame by the time you read this, but I strongly recommend installing version 1.9, since that is the version covered in this book. If a new 2.x has been released, you may have problems compiling the code with a newer version of Pygame. There's nothing wrong with 1.9. New versions of libraries like Pygame are released with new features, not necessarily to fix problems. So, go ahead and download Pygame 1.9.

The website for Pygame is found at http://www.pygame.org.

The latest installer for Pygame 1.9 that supports Python 3.2 is a file called pygame-1.9.2.a0.win32-py3.2.msi. Note that new files will likely be added to the list of installers in the future, but this is the version you want to use for Python 3.2.

There is an alternate site of installers for Pygame and many other Python libraries at this site: http://www.lfd.uci.edu/~gohlke/pythonlibs/. You can scroll down this comprehensive list to find Pygame.

When you run the Pygame installer, it will automatically detect the Python installation folder and will install itself into that location. There are really no options that you will need to change.

CHAPTER B

Pygame Key Codes

Following is a list of all the key codes recognized by Pygame, either with key events or key polling. See Chapter 4 for details on how to use these key codes.

TABLE B.1: PYGAME KEY CODES

Key	ASCII	Common Name
K_BACKSPACE	\b	backspace
K_TAB	\t	tab
K_CLEAR		clear
K_RETURN	\r	return
K_PAUSE		pause
K_ESCAPE	^[	escape
K_SPACE		space
K_EXCLAIM	!	exclaim
K_QUOTEDBL	"	quotedbl
K_HASH	#	hash
K_DOLLAR	$	dollar

TABLE B.1: PYGAME KEY CODES (CONT.)

Key	ASCII	Common Name
K_AMPERSAND	&	ampersand
K_QUOTE	'	quote
K_LEFTPAREN	(	left parenthesis
K_RIGHTPAREN	)	right parenthesis
K_ASTERISK	*	asterisk
K_PLUS	+	plus sign
K_COMMA	,	comma
K_MINUS	-	minus sign
K_PERIOD	.	period
K_SLASH	/	forward slash
K_0	0	0
K_1	1	1
K_2	2	2
K_3	3	3
K_4	4	4
K_5	5	5
K_6	6	6
K_7	7	7
K_8	8	8
K_9	9	9
K_COLON	:	colon
K_SEMICOLON	;	semicolon
K_LESS	<	less-than sign
K_EQUALS	=	equals sign
K_GREATER	>	greater-than sign
K_QUESTION	?	question mark
K_AT	@	at
K_LEFTBRACKET	[	left bracket
K_BACKSLASH	\	backslash
K_RIGHTBRACKET	]	right bracket
K_CARET	^	caret
K_UNDERSCORE	_	underscore
K_BACKQUOTE	`	grave
K_a	a	a
K_b	b	b
K_c	c	c
K_d	d	d

Table B.1: Pygame Key Codes (Cont.)

Key	ASCII	Common Name
K_e	e	e
K_f	f	f
K_g	g	g
K_h	h	h
K_i	i	i
K_j	j	j
K_k	k	k
K_l	l	l
K_m	m	m
K_n	n	n
K_o	o	o
K_p	p	p
K_q	q	q
K_r	r	r
K_s	s	s
K_t	t	t
K_u	u	u
K_v	v	v
K_w	w	w
K_x	x	x
K_y	y	y
K_z	z	z
K_DELETE		delete
K_KP	0	keypad 0
K_KP	1	keypad 1
K_KP	2	keypad 2
K_KP	3	keypad 3
K_KP	4	keypad 4
K_KP	5	keypad 5
K_KP	6	keypad 6
K_KP	7	keypad 7
K_KP	8	keypad 8
K_KP	9	keypad 9
K_KP_PERIOD	.	keypad period
K_KP_DIVIDE	/	keypad divide
K_KP_MULTIPLY	*	keypad multiply
K_KP_MINUS	-	keypad minus

Table B.1: Pygame Key Codes (Cont.)

Key	ASCII	Common Name
K_KP_PLUS	+	keypad plus
K_KP_ENTER	\r	keypad enter
K_KP_EQUALS	=	keypad equals
K_UP		up arrow
K_DOWN		down arrow
K_RIGHT		right arrow
K_LEFT		left arrow
K_INSERT		insert
K_HOME		home
K_END		end
K_PAGEUP		page up
K_PAGEDOWN		page down
K_F1		F1
K_F2		F2
K_F3		F3
K_F4		F4
K_F5		F5
K_F6		F6
K_F7		F7
K_F8		F8
K_F9		F9
K_F10		F10
K_F11		F11
K_F12		F12
K_F13		F13
K_F14		F14
K_F15		F15
K_NUMLOCK		numlock
K_CAPSLOCK		capslock
K_SCROLLOCK		scrollock
K_RSHIFT		right shift
K_LSHIFT		left shift
K_RCTRL		right ctrl
K_LCTRL		left ctrl
K_RALT		right alt
K_LALT		left alt
K_RMETA		right meta

TABLE B.1: PYGAME KEY CODES (CONT.)

Key	ASCII	Common Name
K_LMETA		left meta
K_LSUPER		left windows key
K_RSUPER		right windows key
K_MODE		mode shift
K_HELP		help
K_PRINT		print screen
K_SYSREQ		sysrq
K_BREAK		break
K_MENU		menu
K_POWER		power
K_EURO		euro

This page intentionally left blank

Index

Special Characters

# (pound character), 8
% (modulus character), 87
8-bit Intel processors, 11
16-bit register, 10–11

A

accessor/mutator methods, 138
adaptive programming, 13
all_data variable, 45
American Standard Code for Information Interchange. *See* ASCII
Analog Clock Demo, 83–92
  code for, 89–90
  drawing clock, 84–92
    hours, 86–88
    minutes, 88–89
    numbers, 85–86
    seconds, 89–90
  getting time for, 84
AngBand RPG, 249–250
angular velocity, in Tank Battle Game, 202–206
  calculating, 203
  moving forward and backward at angle, 205–206
  and rotation, 203–205
animation, with sprites, 109–125
  Escape the Dragon Game, 119–125
  frames for, 112–115
  MySprite class, 116–117
  overview, 110–112
  sprite groups, 115
  sprite sheets, 112
  testing, 118–119
append( ) method, 151, 153
arcs, drawing, 30–31
Arnold, Ken, 247
arrays, 150–157
  elements in
    adding, 151
    changing, 151
    counting, 151
    removing, 152
    reversing, 152
    searching for, 152
    sorting, 152
  multi-dimensional, 154–157
    changing many elements in, 155–157
    changing one element in, 154–155
  one-dimensional, 150–153
  queue-like, 153–154
  stack-like, 153
  tuples as constant, 159–160
Artillery Gunner Game, 219–244
  cannons in, 230–235
    drawing turrets, 231–232
    firing, 232–235
    placing, 230–231
    scoring hits, 234–235
  code for, 235–244
  overview, 219–220

Terrain class, 220–230
locating grid points in, 228–230
smoothing terrain, 226–228
ASCII (American Standard Code for Information Interchange), 253–257
character set, 253–254
key code constants, 62
NetHack, 248–249
overview, 245
printing of, 254
table as sprite, 258–259
table program for, 254–257
assembler program, 10
Audacity software, 175
audio. *See* sound
auto_move( ) function, 196
AX register, 10

B

BASIC language, 7, 10
binary files, 45–47
open modes, 46
reading from, 47
writing to, 46–47
bitmap graphics, 94–100
loading, 94–95
in Orbiting Spaceship Demo
drawing background, 95–97
drawing planet, 97–98
drawing spaceship, 98–100
blit( ) function, 95
Block Breaker Game
code for, 169–170
hitting blocks, 168–169
initializing, 165
levels in, 160–164
level 1, 161–162
level 2, 162
level 3, 163
loading and changing, 163–164
moving ball, 166–167
MySprite class update, 170–171
paddle in
hitting, 167–168
moving, 165–166
block_group sprite group, 164
blocks.png file, 170
Bomb Catching Game, 67–71
bounding circle technique, 121
bounding rectangle technique, 121
Bug class, 15
Bullet class, 211–212, 214
Bullet.owner identifier, 214

C

C language, 8
CalcArea( ) method, 20
CalcCircum( ) method, 20
cannons, in Artillery Gunner Game, 230–235
drawing turrets, 231–232
firing, 232–235
placing, 230–231
scoring hit, 234–235
Cartesian coordinate system, 79
cathode ray tube (CRT) monitor, 257
Channel class, 175
Channel.play( ) method, 175
chr( ) function, 62, 254–255
Circle class, 16–17
circle collisions, 129
Circle Demo, 81–83
circles
drawing, 25–26
theory for, 75–79
traversing perimeter, 79–81
calculating X, 79–81
calculating Y, 81
class definition, 14

classes, 14
Clock( ) method, 113
collision detection
  circle collisions, 129
  masked collisions, 130
  rectangle collisions
    between sprite and group, 130–131
    between two groups, 131
    between two sprites, 128–129
  Zombie Mob Game
    code for, 141–146
    custom modules for, 132–135
    directional animation, 135–138
    overview, 127–128
collision_ball_blocks( ) function, 168
collision_ball_paddle( ) function, 167
colon, 14
color alpha channel manipulation, 173
command prompt, 4–6
compilers, 12–13
computer_firing variable, 232
Conley, Ronald, 98
convert( ) function, 95
convert_alpha( ) function, 95
cosine math function, 74–75
count( ) method, 151, 159
counting elements
  in lists and arrays, 151
  in tuples, 159
CRT (cathode ray tube) monitor, 257

D

darktan color, 178
data types, 36–43
  handling exceptions for, 40
  and input( ) function, 39–40
  Mad Lib Game, 40–43
and print( ) function, 37–39
date module, 84
datetime class, 39
datetime.today( ) function, 84
Dawson, Michael, 2
def __init__(self) line, 14
def word, 8
Defense value, 292
degrees, 75
delta_x parameter, 103
delta_y parameter, 103
detection, collision. *See* collision detection
device polling, 61–67
  keyboard, 61–65
  mouse, 65–67
Die function, 283
distance( ) function, 234
draw( ) method, 112, 115
draw_computer_cannon( ) function, 232
drawing
  arcs, 30–31
  circles, 25–26
  lines, 28–29
  rectangles, 27–28
Drawing Rectangles Demo, 34
draw_player_cannon( ) function, 231
Dungeon class, 263, 287
Dungeon RPG. *See* RPGs
Dungeon.createHallRight( ) method, 267
Dungeon.draw( ) method, 297
Dungeon.draw_radius( ) method, 297
Dungeon.entrance_x variable, 283
Dungeon.entrance_y variable, 283
Dungeon.generate( ) method, 261
Dungeon.py file, 278, 287

E

elements
  in lists and arrays
    adding, 151
    changing, 151

counting, 151
removing, 152
reversing, 152
searching for, 152
sorting, 152
in tuples
counting, 159
searching for, 159
elif statement, 60
Ellipse class, 20
encapsulation, in Python, 15
EnemyTank class, 207
entities, 13
EOP (Entity Oriented Programming), 13
Epic Games company, 251
ErrorProne( ) function, 12
Escape key, 60, 62
Escape the Dragon Game, 119–125
code for, 122–125
colliding, 121–122
jumping, 120–121
event.button property, 61
event.buttons property, 61
event.pos property, 61
event.rel property, 61
events, for Pygame library, 58–61
keyboard events, 60–61
mouse events, 61
real-time loop, 59–60
except block, 40
exceptions, handling, 40
exiting level, in Dungeon RPG, 299

F

Feldman, Ari, 111, 208
FIFO (first-in, first-out) mechanism, 153
file input/output, 43–47
for binary files, 45–47
reading from, 47
writing to, 46–47
for text files, 43–45
reading from, 44–45
writing to, 44
File menu
New Window option, 4–5
saving file, 6
file.readline(n) function, 45
file.readlines( ) function, 45, 48
file.read(n) function, 44
file.write( ) function, 44
file.writelines( ) fuction, 44
FILO (first-in, last-out) mechanism, 153
find_channel( ) method, 175
firing cannons, in Artillery Gunner Game, 232–235
first-in, first-out (FIFO) mechanism, 153
first-in, last-out (FILO) mechanism, 153
float( ) function, 37, 40
floating-point velocity value, 68
font.render( ) function, 48
food_group sprite group, 188, 193
for loop, 59–60, 155, 160
format( ) function, 80
FORTRAN language, 10
framerate variable, 113
frames, for sprites
changing, 112–114
drawing one, 114–115
function definition, 9
functions, 8

G

game_init( ) function, 189, 191, 212, 221
games. *See also* RPGs
Artillery Gunner, 219–244
cannons in, 230–235
code for, 235–244
overview, 219–220

Terrain class, 220–230
Block Breaker
code for, 169–170
hitting blocks, 168–169
initializing, 165
levels in, 160–164
moving ball, 166–167
MySprite class update, 170–171
paddle in, 165–168
Bomb Catching Game, 67–71
Epic Games company, 251
Escape the Dragon, 119–125
code for, 122–125
colliding, 121–122
jumping, 120–121
game_init( ) function, 189, 191, 212, 221
Mad Lib, 40–43
Oil Spill, 176–179
code for, 179–182
oil, 176–178
timing in, 176
washing background, 178–179
Pygame library, 21–34
events for, 58–61
installing, 309–310
key codes for, 311–315
overview, 22, 30–31
reference manual, 23
using, 22–31
Roguelike games, 246–247
RTS games, 79
Snake, 185–199
biting self, 194–195
falling off world, 195
Food class, 188–189
growth by eating food, 193–194
initializing, 189–191
moving snake automatically, 195–199
program main, 191–193
Snake class, 187–188
SnakeSegment class, 187
SpriteLib sprites, 111
Starflight—The Lost Colony game, 98
Tank Battle, 201–218
angular velocity in, 202–206
Bullet class, 211–212
calculating, 203
main code, 212–218
moving forward and backward at angle, 205–206
overview, 201–202
and rotation, 203–205
Tank class, 207–211
Trivia, 47–56
code for, 53
displaying question and answers, 50–52
going to next question, 53
loading trivia data for, 49–50
printing text in, 48
responding to user input, 52–53
Trivia class for, 48–49
Video Game Programming for Kids book, 7
Visual C# Game Programming for Teens book, 297
Zombie Mob, 127–128
code for, 141–146
custom modules for, 132–135
directional animation, 135–138
generate( ) method, 226, 265
GeometryDemo.py program, 20
get( ) method, 138
getCharAt( ) method, 260, 271
get_current_direction( ) function, 197
get_food_direction( ) function, 197
get_ticks( ) method, 113
global keyword, 164
gosub command, 9
goto command, 9

goto_next_level( ) function, 163
grid_point variable, 229
grid_size property, 228

H

hallways, in Dungeon RPG, 266–272
  handling range errors, 271–272
  horizontal, 266–269
  vertical, 269–271
handling exceptions, 40
health sprite, 139–140
hour property, 86

I

IDLE editor, 2, 7, 255
Idle, Eric, 2
IDLE (Python GUI), 4–6
image property, 110
import statement, 46
in sequence operator, 159
index( ) method, 152
inheritance
  multiple inheritance, 16–17
  in Python, 15–16
  single inheritance, 16–17
__init__( ) function, 48–49
input( ) function, 39–40
insert( ) method, 151
installing
  Pygame, 309–310
  Python, 305–309
int( ) function, 37
integer division, 99

J

joystick input, 59

K

key code constants, ASCII code, 62
Keyboard Demo, 62
keyboard events, for Pygame library, 60–61
keyboard polling, 61–65
KEYDOWN event, 60
key.name property, 60
KEYUP event, 60
Kingdom of Kroz RPG, 251

L

Lane, Jon, 247
language, of Python, 7
len( ) function, 159–160
lines, drawing, 28–29
list class, 149, 258
lists, 150–157
  elements in
    adding, 151
    changing, 151
    counting, 151
    removing, 152
    reversing, 152
    searching for, 152
    sorting, 152
  multi-dimensional, 154–157
    changing many elements in, 155–157
    changing one element in, 154–155
  one-dimensional, 150–153
  queue-like, 153–154
  stack-like, 153
load( ) method, 112, 170–171
loading
  data, for Trivia Game, 49–50
files, for sounds, 174–175
load_level( ) function, 163–164
loat_pos property, 206
logic, programming in Snake Game, 185–199
  biting self, 194–195
  falling off world, 195
  Food class, 188–189

growth by eating food, 193–194
initializing, 189–191
moving snake automatically, 195–199
program main, 191–193
Snake class, 187–188
SnakeSegment class, 187
looping, 24–25
LUA language, 11

M

Mad Lib Game, 40–43
mask property, 130
masked collisions, 130
master_image image, 170
math module, 75
math.atan2( ) function, 103, 202
math.cos( ) function, 79–80
math.degrees( ) function, 78, 202
math.radians( ) function, 30–31, 78
math.sin(angle) function, 81
message( ) function, 290, 300
methods, 8, 14
mnemonic programming, 10–11
Module Docs (Pydoc), 3
modulus character (%), 87
Monster class, 283, 287, 296
monsters, in Dungeon RPG
creating, 277–278
fighting, 291–295
hitting, 295–296
Monty Python, 2
Mouse Demo, 65
mouse events, for Pygame library, 61
mouse polling, 65–67
MOUSEBUTTONDOWN event, 61
MOUSEBUTTONUP event, 61
MOUSEMOTION event, 61, 228
move_ball( ) function, 166
move_monster( ) function, 296
move_monsters( ) function, 296
MP3 audio files, 174
multi-dimensional arrays, 154–157
changing many elements in, 155–157
changing one element in, 154–155
multi-dimensional lists, 154–157
changing many elements in, 155–157
changing one element in, 154–155
multiple inheritance, 15–17
MyLibrary.py file, 132, 141, 160, 170, 206, 233, 235
MySprite class, 116–117, 119, 129, 132, 137, 170–171, 205
MySprite object, 258
MySprite.direction property, 138

N

\n character, 37, 44
NetHack RPG, 248–249
new source code editor window, 4–6
New Window option, File menu, 4–5
non-indented line, 8
non-printing characters, 253

O

object-oriented programming. *See* OOP
objects, 14
OGG audio files, 174–175
Oil Spill Game, 176–179
code for, 179–182
oil
cleaning, 177–178
mess, 176–177
timing in, 176
washing background, 178–179
one-dimensional arrays, 150–153
one-dimensional lists, 150–153
OOP (object-oriented programming), 2, 7–20
prior methodologies, 7–11

mnemonic programming, 10–11
sequential programming, 9–10
structured programming, 8–9
in Python, 13–16
encapsulation, 15
inheritance, 15–16
polymorphism, 14–15
open( ) function, 43
open modes
binary file, 46
text file, 43
OpenGL, 10
Orbiting Spaceship Demo
planet in, 101–107
orbiting, 101–103
rotating, 103–107
using bitmaps in
drawing background, 95–97
drawing planet, 97–98
drawing spaceship, 98–100

P

packing tuples, 158
paddle, in Block Breaker Game
hitting, 167–168
moving, 165–166
PI number, 75
Pie Game. *See* Pygame library
planet, in Orbiting Spaceship Demo, 101–107
orbiting, 101–103
rotating, 103–107
play( ) method, 175
Player class, 283, 287, 291
player_cannon_position variable, 232
player_cannon_power variable, 232
playerCollision( ) function, 287–288
player_firing variable, 232
Player.MoveDown( ) method, 287
Player.MoveUp( ) method, 287
Player.py file, 283, 287
player_shell_position variable, 232
player.update( ) function, 138
playing audio, 175
PNG (Portable Network Graphics) bitmap file, 45
Point class, 16–18, 101–102, 132, 203
Point.__init__( ) method, 19
Point.ToString( ) method, 19
polymorphism, in Python, 14–15
pop( ) method, 153
Portable Network Graphics (PNG) bitmap file, 45
pound character (#), 8
print( ) function, 4, 9, 12, 37–39, 48, 254
printing text, 23–24
PrintName( ) function, 9
print_text( ) function, 54, 132
procedural programming, 8
procedures, 8
program logic, Snake Game, 185–199
biting self, 194–195
falling off world, 195
Food class, 188–189
growth by eating food, 193–194
initializing, 189–191
moving snake automatically, 195–199
program main, 189–191
Snake class, 187–188
SnakeSegment class, 187
property( ) function, 101
putCharInRandomRoom( ) method, 276
Pydoc (Module Docs), 3
Pygame library, 21–34
events for, 58–61
keyboard events, 60–61
mouse events, 61
real-time loop, 59–60
installing, 309–310

key codes for, 311–315
overview, 22, 30–31
reference manual, 23
using, 22–31
drawing arcs, 30–31
drawing circles, 25–26
drawing lines, 28–29
drawing rectangles, 27–28
looping, 24–25
printing text, 23–24
pygame1.9.2.a0.win32-py3.2.msi file, 309
pygame.display.get_surface( ) function, 132
pygame.display.set_mode( ) method, 93–94, 190
pygame.draw.arc( ) function, 30–31
pygame.draw.circle( ) function, 25–26
pygame.draw.line( ) function, 28–29, 87
pygame.draw.rect( ) function, 27–28
pygame.font module, 48
pygame.font.Font( ) constructor, 23, 48
pygame.gfxdraw module, 95
pygame.gfxdraw.pixel( ) function, 95
pygame.image class, 93
pygame.image.load( ) function, 94
pygame.key,get_pressed( ) method, 61
pygame.mixer module, 174
pygame.mixer.Channel class, 175
pygame.mixer.Sound( ) class, 174–175
pygame.mouse.get_pos( ) function, 65
pygame.mouse.get_pressed( ) function, 65
pygame.mouse.get_rel( ) function, 65
pygame.sprite module, 110
pygame.sprite.collide_circle( ) function, 129
pygame.sprite.collide_circle_ratio( ) function, 129, 176, 178
pygame.sprite.collide_mask( ) function, 130
pygame.sprite.collide_rect( ) function, 128
pygame.sprite.collide_rect_ratio( ) function, 129, 138
pygame.sprite.collide_rest( ) method, 122
pygame.sprite.Group class, 115, 176, 193
pygame.sprite.groupcollide( ) function, 131, 193
pygame.sprite.Sprite class, 98, 122, 128, 176, 205–206
pygame.sprite.spritecollide( ) function, 130
pygame.sprite.spritecollideany( ) function, 130–131, 138, 168
pygame.Surface class, 93
pygame.time module, 113
pygame.time.get_ticks( ) method, 176
pygame.transform module, 99, 104
pygame.transform.rotate( ) function, 104
pygame.transform.scale( ) function, 99
pygame.transform.smoothscale( ) function, 99
Pyhelp (Python Manuals), 4
Python
command line, 4
documentation tool, 3
installing, 305–309
software package, 2
Python GUI (IDLE), 4–6
Python Manuals (Pyhelp), 4
Python Programming for the Absolute Beginner book, 2
Python Shell, 6

Q

QB64 tool, 7, 10
QBASIC language, 10
queue-like arrays, 153–154
queue-like lists, 153–154
QuickBasic language, 10
QUIT event, 60

R

radians, 75
radius property, 129

random.randint( ) function, 34, 64
rangle variable, 104
reading
  from binary files, 47
  from text files, 44–45
real-time strategy (RTS) games, 79
rect property, 110, 116, 122
Rectangle class, 18–19
rectangle collisions
  between sprite and group, 130–131
  between two groups, 131
  between two sprites, 128–129
rectangles, drawing, 27–28
remove( ) method, 152
removing elements, in lists and arrays, 152
reserved words, 12
reset_ball( ) function, 166
ret property, 128
reverse( ) method, 152
reversing elements, in lists and arrays, 152
Ritchie, Dennis, 248
Rogue RPG, 247–248
Roguelike games, 246–247
Romero, George A., 146
rooms, in Dungeon RPG, 260–265
  Dungeon class, 260–261
  northern, 261–263
  southern, 263–265
RPGs (role-playing games), 245–304
  and ASCII characters, 253–257
    character set, 253–254
    printing of, 254
    table program for, 254–257
  classics, 246–252
    AngBand, 249–250
    Kingdom of Kroz, 251
    NetHack, 248–249
    Rogue, 247–248
    Sword of Fargoal, 251
    ZZT, 251–252
  exiting level, 299
  hallways in, 266–272
    handling range errors, 271–272
    horizontal, 266–269
    vertical, 269–271
  message( ) function in, 300
  monsters
    creating, 277–278
    fighting, 291–295
    hitting, 295–296
  overview, 246
  picking up items, 288–291
  player character, 283–287
    moving, 286–287
    Player class, 283–286
  populating dungeon, 272–287
    with armor, 276–277
    entrance and exit portals, 273–274
    with gold, 274–275
    with health potions, 276–277
    with weapons, 276–277
  printing game stats, 299–300
  rooms in, 260–265
    Dungeon class, 260–261
    northern, 261–263
    southern, 263–265
  text console display for, 257–260
    ASCII table as sprite, 258–259
    drawing, 259–260
    keeping track of tiles, 258
  visibility range, 297–298
RTS (real-time strategy) games, 79
Run menu, 6
Run Module option, 6

S

scratch_ship image, 104
screen object, 95

screen.blit( ) function, 23, 48
scripting languages, 11
SDKs (software development kits), 8
SDL (Simple DirectMedia Layer)
  drawing functions, 95
  library, 57
searching elements
  in lists and arrays, 152
  in tuples, 159
self key word, 14
self.failed flag, 52
self.scored flag, 52
sequential programming, 9–10
set( ) method, 138
setCharAt( ) method, 260, 271
set_image( ) method, 171
shortcuts, F5 key, 6
Simple DirectMedia Layer. *See* SDL
sine math function, 74–75
single inheritance, 16–17
Size class, 18–19
Size.__init__( ) method, 19
Size.ToString( ) method, 19
Snake class, 193, 198
Snake Game, 185–199
  biting self, 194–195
  falling off world, 195
  Food class, 188–189
  growth by eating food, 193–194
  initializing, 189–191
  moving snake automatically, 195–199
  program main, 189–191
  Snake class, 187–188
  SnakeSegment class, 187
snake.segments[0] sprite, 194
software development kits (SDKs), 8
sort( ) method, 152
sorting, elements, in lists and arrays, 152
sound, 174–175
  loading files, 174–175
  playing audio, 175
Sound class, 175
Space key, 60
Space Shuttle program, 55
spaghetti code, 10
Sprite class, 110, 112, 130, 137
sprite groups, 115
sprite sheets, 112, 136–137
Sprite.draw( ) method, 114
Sprite.image property, 114
SpriteLib sprites, 111
Sprite.rect property, 114–115
sprites, 109–125. *See also* animation, with sprites;
  collision detection
  Escape the Dragon Game, 119–125
    code for, 122–125
    colliding, 121–122
    jumping, 120–121
  frames for
    changing, 112–114
    drawing one, 114–115
  MySprite class, 116–117
  overview, 110–112
  sprite groups, 115
  sprite sheets, 112
  testing, 118–119
stack-like arrays, 153
stack-like lists, 153
Starflight—The Lost Colony game, 98
starting point, 110
state-based programming, 32
stop( ) function, 175
str( ) function, 37, 210
string.format( ) function, 80
string.replace( ) function, 41
string.strip( ) function, 45
struct library module, 46

struct.calcsize( ) function, 47
struct.pack( ) function, 46
struct.unpack( ) function, 46–47
structured programming, 8–9
subsurface( ) method, 114–115
super( ) method, 16–17
Surface class, 93–95, 97
Surface.convert_alpha( ) function, 95
Surface.get_at( ) function, 130
Surface.get_height( ) function, 97
Surface.get_size( ) function, 97, 104
Surface.get_width( ) function, 97
Surface.lock( ) function, 130
Surface.set_at( ) function, 130
Surface.unlock( ) function, 130
Sweeney, Tim, 251
Sword of Fargoal RPG, 251
sys.copyright value, 38
sys.exit( ) function, 59
sys.platform value, 38

T

T variable, 84
tan color, 178
tangent math function, 101
Tank Battle Game, 201–218
  angular velocity in, 202–206
    calculating, 203
    moving forward and backward at angle, 205–206
    and rotation, 203–205
  Bullet class, 211–212
  main code, 212–218
    audio functions, 213–214
    game initialization, 212–213
    gameplay code, 214
    header code, 212
  overview, 201–202
  Tank class, 205–211
    constructor, 207
    draw function, 209–210
    and EnemyTank class, 210–211
    string override for, 210
    update function, 207–209
Tank class, 205–211
  constructor, 207
  draw function, 209–210
  and EnemyTank class, 210–211
  string override for, 210
  update function, 207–209
Tank.update( ) function, 207
target_angle( ) function, 205
Terrain class, in Artillery Gunner Game, 220–230
  locating grid points in, 228–230
  smoothing terrain, 226–228
Terrain.draw( ) method, 222, 229
testing sprites, 118–119
text files, 43–45
  open modes, 43
  reading from, 44–45
  writing to, 44
text, printing, 23–24
textImage object, 23
T.hour property, 84
ticks variable, 113
time, getting for Analog Clock Demo, 84
time interval, 113
time module, 64, 84
time.clock( ) function, 64–65
T.microsecond property, 84
T.minute property, 84
tools and utilities, 3–6
  IDLE (Python GUI), 4–6
  Module Docs (Pydoc), 3
  Python (command line), 4
  Python Manuals (Pyhelp), 4
ToString( ) method, 17–19

total_points parameter, 228
Toy, Michael, 247
trigonometry, 74–83
  Circle Demo, 81–83
  circle theory, 75–79
  traversing circle perimeter, 79–81
    calculating X, 79–81
    calculating Y, 81
triple quotes, 39
Trivia class, 48–49, 54
Trivia Game, 47–56
  code for, 53
  displaying question and answers, 50–52
  going to next question, 53
  loading trivia data for, 49–50
  printing text in, 48
  responding to user input, 52–53
  Trivia class for, 48–49
trivia object, 54
Trivia.current field, 50, 53
Trivia.data list, 49
trivia_data.txt file, 49, 54
Trivia.handle_input( ) method, 51
Trivia.next_question( ) method, 53
Trivia.show_question( ) method, 50, 54
try: block, 40
try...except block, 40
T.second property, 84
tuples, 157–160
  as constant arrays, 159–160
  elements in, 159
  packing, 158
  unpacking, 158

U

UDTs (user-defined types), 8
Unicode characters, 255
UNIX operating system, 8
unpacking tuples, 158
update( ) method, 112–115, 171, 198
update_blocks( ) function, 163
user-defined types (UDTs), 8
utilities. *See* tools and utilities

V

velocity, in Tank Battle Game, 202–206
  calculating, 203
  moving forward and backward at angle, 205–206
  and rotation, 203–205
VGA mode 13h, 10
Video Game Programming for Kids book, 7
Visual C# Game Programming for Teens book, 297

W

waiting flag, 166
Walk( ) method, 15
WAV audio files, 174–175
while block, for Pygame library, 59–60
while keyword, 24
while loop, 24, 27, 191
while statement, 59
Wichman, Glenn, 247
Windows help file format, 4
Windows x86-64 version, installing Python, 305
wrap_angle( ) function, 88, 206
write property, 44
writing
  to binary files, 46–47
  to text files, 44

X

X axis, 79
X property, 101

Y

Y axis, 79

Y property, 101

Z

Zombie Mob Game, 127–128
    code for, 141–146
    custom modules for, 132–135
    directional animation, 135–138
ZZT RPG, 251–252