Beginning\_Game\_Development\_Python\_Pygame\_c02

CHAPTER 2

■■■

Exploring Python

I

n the previous chapter we entered our Python code a line at a time, but now we are going to

put the interactive interpreter to the side and start creating Python files. In this chapter we will

cover more of the building blocks of Python code, and show you how to use classes to help with

creating games. We will also explain how to use the code libraries that come with all installa-

tions of Python.

Creating Scripts

A file containing Python code is called a script. All you need to create scripts is a simple text edi-

tor, but it’s best to use a Python-aware editor such as SciTE (see Figure 2-1) or IDLE (which

comes with the standard distribution of Python).

Just save your script with a .py extension, so that your operating system knows that it con-

tains Python. To run a script, you typically just double-click it, or if you prefer the command

line, type python followed by a space and the name of your script. Most Python editors will have

a shortcut key to run the script you have been editing.

■Note Python allows both spaces and tabs for indentation, but it is better to use four spaces. Most Python

editors will be set this way, but you may have to check it in the Options dialog of your editor.

1920

CHAPTER 2 ■ EXPLORING PYTHON

Figure 2-1. Editing Python code with SciTE (www.scintilla.org/SciTE.html)

Working with Logic

Not just for green-blooded Vulcans, logic is an important part of any software—and that

includes games. Any game needs to make decisions based on information it either has been

given or has calculated. If a laser blast hits the player’s hover tank, the game has to decide if

enough damage has been done to destroy it—and display an explosion animation if it has. This

is just an example of a whole host of decisions a computer game has to make in order to con-

vince us that it is more than just a dumb machine. So please put on your Spock ears and we will

cover logic.

Understanding Booleans

The logic that computers use is boolean logic, so called because it was invented by George

Boole way back in the 1800s—a few years before PlayStation hit the street.

You have already seen logic used in the previous chapter as part of a while loop; count<=5

is a logical expression, and like all logical expressions it results in either True or False. These

truth values, as they are called, are used when making decisions. In the case of a while loop, a

value of True tells Python to carry on around the loop one more time, but a value of FalseCHAPTER 2 ■ EXPLORING PYTHON

causes Python to skip past the code block. (See Listing 2-1 for a few examples of logical expres-

sions.) The important thing to remember with boolean logic is that there are no in-between

values: you can’t have 25 percent true and 75 percent false—it’s always one or the other!

Listing 2-1. Simple Logic

score = 100

health = 60

damage = 50

fugu = "tasty"

print score != 100

print health - damage > 0

print fugu == "tasty"

Running this simple script produces the following output:

False

True

True

Boolean values can be treated just like any other Python type, so you could have a variable

that refers to a boolean. For example, if we were to add the line is\_fugu\_tasty = (fugu ==

"tasty"), then is\_fugu\_tasty would refer to the value True.

If Statement

Logical expressions come into their own when using if statements. You use if statements to

run code only when a condition is true. If the condition is false, then Python jumps past the end

of your code block. Here is a simple example of the if statement:

if fugu == "tasty":

print "Eat the fugu!"

The condition uses the comparison operator (==) to compare a variable with a string.

Assuming we are using the value of fugu from Listing 2-1, this comparison will result in True,

which gives Python the green light to run the indented code block.

And Operator

Often you have several conditions you want to check for. Let’s say we want to eat fugu if it is

tasty and under $100 a plate. We can combine these two conditions with the and operator:

price = 50.0

if fugu == "tasty" and price < 100.0:

print "Eat the fugu!"

2122

CHAPTER 2 ■ EXPLORING PYTHON

Here we will only eat the fugu if fugu is set to "tasty" and the price has a value less than

100.0. Table 2-1 lists how the and operator combines values, but I hope it is self-explanatory.

You can find this operator at work in real life; for example, my car will start only if the battery is

not dead and I have gas.

Table 2-1. Truth Table for the And Operator

LogicResult

False and FalseFalse

True and FalseFalse

False and TrueFalse

True and TrueTrue

Or Operator

To complement the and operator, we have the or operator, which results in True if either the

first or second value is True. Let’s say we want to eat fugu if it is tasty or it is very cheap (after all,

who can turn down cheap fugu?):

if fugu == "tasty" or price < 20.0:

print "Eat the fugu!"

Just like the and operator, the or operator has plenty of applications in real life. My car will

stop if I run out of gas or the battery goes dead. See Table 2-2 for the or truth table.

Table 2-2. Truth Table for the Or Operator

LogicResult

False or FalseFalse

True or FalseTrue

False or TrueTrue

True or TrueTrue

Not Operator

The final logic operator we will look at is the not operator, which swaps the state of a boolean

value, so True becomes False and False becomes True (see Table 2-3). You can use this to

reverse any condition:

if not (fugu == "tasty" and price < 100.0):

print "Don't eat the fugu!"

Here we have reversed the fugu == "tasty" and price < 100.0 condition so that Python runs

the code block only if the condition is not true (that is, false).CHAPTER 2 ■ EXPLORING PYTHON

Table 2-3. Truth Table for the Not Operator

LogicResult

not TrueFalse

not FalseTrue

Else Statement

You may have noticed that the previous snippet is the opposite of our first fugu logic statement.

We have an action that occurs when a condition is true and another that runs when that same

condition is not true. This is such a common situation that Python has a way of tacking on the

alternative action to an if statement. The else statement follows an if statement and intro-

duces a new code block that will run only if the condition is false. Let’s see an example of how

else is used:

if fugu == "tasty":

print "Eat the fugu!"

else:

print "Don't eat the fugu!"

When Python runs this code, it will run the first print statement if the condition is true; else it

will run the second condition.

Elif Statement

Often another if statement will follow an else statement. Python combines an else followed

by an if into a single statement: the elif statement. Let’s say we want to classify the fugu into

three categories based on the price. For sake of argument, we will classify $20–$100 as reason-

ably priced fugu, anything above that range as expensive fugu, and anything below that as

cheap fugu. Python can do this for us using elif:

if price < 20:

print "Cheap fugu!"

elif price < 100:

print "Reasonably priced fugu."

else:

print "Expensive fugu!"

Here we have three code blocks, but only one of them runs. If price is less than 20, then the

first block will run; if price is less than 100, then the second block will run; and any other value

of price will cause the third block to run. You can have as many elif statements as you want

after an if, but if you have an else statement it must come at the end.

Understanding Functions

A function is a stored piece of Python code that you can pass information to and potentially get

information back from. Python provides a large number of useful functions (see Table 2-4 for

some examples), but you can also create your own.

2324

CHAPTER 2 ■ EXPLORING PYTHON

Table 2-4. Some Built-in Python Functions

FunctionDescriptionExample

absFinds the absolute value of a numberabs(-3)

helpDisplays usage information for any Python objecthelp([])

lenReturns the length of a string or collectionlen("hello")

maxReturns the largest valuemax(3, 5)

minReturns the smallest valuemin(3, 4)

rangeReturns a list containing a range of numbersrange(1,6)

roundRounds a float to a given precisionround(10.2756, 2)

\* For a more comprehensive list of Python’s built-in functions, see the Python documenta-

tion, or visit http://doc.python.org.

Defining Functions

To define a function in Python, you use the def statement followed by the name you want to

give to your function. You can use any name you want, but it is a good idea to give it a name

that describes what it actually does! Function names are typically in lowercase and may use

underscores to divide words. Listing 2-2 is a simple Python function for calculating the tip on a

meal of fugu.

Listing 2-2. Calculating the Tip on Fugu

def fugu\_tip(price, num\_plates, tip):

total = price \* num\_plates

tip = total \* (tip / 100.)

return tip

print fugu\_tip(100.0, 2, 15.0)

print fugu\_tip(50.0, 1, 5.0)

This script produces the following output:

30.0

2.5

When Python first encounters a def statement, it knows to expect a function definition,

which consists of the name of the function followed by a list of parameters in parentheses. Just

as with for, while, and if statements, a colon is used to introduce a block of code (known as the

function body). In the aforementioned statements, the block of code doesn’t run immedi-

ately—it’s just stored away until it is needed. Calling the function causes Python to jump to the

beginning of the function body and assign information given in the call to each of the parame-

ters. So in Listing 2-2, the first call to fugu\_tip runs with price set to 100, num\_plates set to 2,

and tip set to 15.CHAPTER 2 ■ EXPLORING PYTHON

The only thing you haven’t already encountered in fugu\_tip is the return statement,

which tells Python to jump back from the function, potentially with some new information. In

the case of fugu\_tip we return with the value of our tip, but a function can return any Python

object.

■Note You don’t need a return statement in the function. Without a return statement, a function will

return when it gets to the end of the code block with the value None—which is a special Python value indi-

cating “nothing here.”

You may have noticed that inside fugu\_tip two variables are created; these are called local

variables, because they only exist inside the function. When the function returns, total and tip

will no longer exist in Python’s memory—although it is possible to have variables with the

same name outside of the function.

Default Values

Parameters can have a default value, which is used if you don’t supply a value in the function

call. Without default values Python will throw an exception if you forget a parameter. Let’s give

default values to fugu\_tip. I am a generous tipper, so we’ll set the default of tip to be 15 (which

represents a percentage of the meal cost), and since I don’t like to eat alone we will assign

num\_plates a default of 2.

To set default values in Python, append the parameter name with a = symbol followed by

the value you want to give it. See Listing 2-3 for a modified fugu\_tip function with these default

values. fugu\_tip can now be called with just one value; the other two values are filled in auto-

matically if you omit them. There can be as many default values in a function definition as you

want, but parameters with defaults must come at the end of the parameter list.

Listing 2-3. Calculating the Tip on Fugu

def fugu\_tip(price, num\_plates=2, tip=15.):

total = price \* num\_plates

tip = total \* (tip / 100.)

return tip

print fugu\_tip(100.0)

print fugu\_tip(50.0, 1, 5.0)

print fugu\_tip(50.0, tip=10.0)

Running this code gives us the following values for tips:

30.0

2.5

10.0

2526

CHAPTER 2 ■ EXPLORING PYTHON

You may have noticed that there is something unusual about Listing 2-3. The third call

to fugu\_tip omits a value for num\_plates and sets the value of tip by name. When you set

parameters explicitly like this, they are called keyword arguments. They are useful if you have

a function with many parameters, but you only need to set a few of them. Without defaults,

parameters must be given in the same order as the parameter list.

Introducing Object-Oriented Programming

You may have heard the term object-oriented programming (OOP) before. But don’t worry if

you are unfamiliar with it, because the concept is remarkably simple.

So what exactly is an object in OOP terms? Well, it can literally be anything. In a game we

may have an object for a particle—say, a burning ember emitted from an explosion, or the

hover tank that caused the explosion. In fact, the entire game world could be an object. The

purpose of an object is to contain information and to give the programmer the ability to do

things with that information.

When constructing an object, it is usually best to start by working out what information, or

properties, it contains. Let’s think about what would be found in an object designed to repre-

sent a futuristic hover tank. It should contain a bare minimum of the following properties:

• Position: Where is the tank?

• Direction: In what direction is it facing?

• Speed: How fast is it going?

• Armor: How much armor does it have?

• Ammo: How many shells does it have?

Now that we have the information to describe a tank and what it is doing, we need to give

it the ability to perform all the actions that a tank needs to do in a game. In OOP-speak, these

actions are called methods. I can think of the following methods that a tank will definitely

require, but there will likely be far more:

• Move: Move the tank forward.

• Turn: Rotate the tank left/right.

• Fire: Launch a shell.

• Hit: This is the action when an enemy shell hits the tank.

• Explode: Replace the tank with an explosion animation.

You can see that the methods typically change the properties of an object. When the Move

method is used, it will update the tank’s Position. Similarly, when the Fire method is used, the

value of Ammo will be updated (unless of course there is no Ammo left; then Fire would not do

anything!).CHAPTER 2 ■ EXPLORING PYTHON

Using Classes

A class is Python’s way of defining objects. You have actually used classes before; lists, dictio-

naries, and even strings are all classes, but you can also create your own. Think of a class as a

kind of a template for an object, because you define the class once and use it to create as many

objects as you need. Let’s write a simple Tank class (Listing 2-4); we will use it later to create a

simple game.

Listing 2-4. An Example Tank Class Definition

class Tank(object):

def \_\_init\_\_(self, name):

self.name = name

self.alive = True

self.ammo = 5

self.armor = 60

When Python encounters class Tank(object):, it creates a class called Tank, which is

derived from the base class called object. Deriving from a class means building on what it does.

We could first create a class called Vehicle, which could handle moving and turning, and then

create a tank by deriving from it and adding the ability to fire weapons. The advantage of this

approach is that Vehicle could be reused to give other game entities the ability to rotate and

move. For this example, we don’t have another class to build on, so our base class will be

object, which is a simple class built into Python itself.

■Note I may have given you the impression that object doesn’t do much, but actually it does a lot of use-

ful things behind the scenes when working with classes—you just don’t use it directly.

Everything in the indented code block after the class statement is the class definition. This

is where we set up the properties used to describe the object and supply all the methods that it

will need. In Python, properties are simply variables stored within the object, and methods are

functions that work with the object. In our Tank class, there is an oddly named method called

\_\_init\_\_, which has special meaning to Python. When you create an object, Python calls this

method automatically. It is typically used by Python programmers to assign properties to the

object, but you can do anything else that may be needed when the object is first created.

This \_\_init\_\_ method takes two parameters: self and name. Because methods are poten-

tially used for many objects, we need some way to know which object we are using. This is

where self comes in—it is a reference to the current object that Python supplies automatically

to all method calls. The second parameter (name) is a string we will use to tell one tank from

another, because there will be more than one.

2728

CHAPTER 2 ■ EXPLORING PYTHON

The code in \_\_init\_\_ first copies the name parameter to a property so we can retrieve it

later; it then assigns a few other properties we will need. We don’t require a great deal of infor-

mation for the tank in the game I have planned; we just need to know if the tank is alive

(self.alive), how much ammo it has (self.ammo), and how much armor it has remaining

(self.armor).

■Note You don’t have to call the first parameter self. You could name it anything you want, but it is a good

idea to stick to self so you will know exactly what it is for when you read over your code. Python program-

mers tend to stick to this convention, so there is no confusion when exchanging code.

Now that we have a tank definition, we can create a new tank by calling Tank, which we

supply with a string. Let’s see an example:

my\_tank = Tank("Bob")

This creates a new tank called Bob and calls \_\_init\_\_ to initialize it. Bob the tank is then

assigned to the variable my\_tank, which is called an instance of the Tank class. We can now treat

my\_tank as an individual object—passing it into functions, storing it in lists, and so forth, or we

can access the properties individually. For instance, print my\_tank.name would display Bob.

With just one method, the Tank class can’t do anything interesting. Let’s flesh it out with a

few more methods in Listing 2-5.

Listing 2-5. Extended Tank Class

def \_\_str\_\_(self):

if self.alive:

return "%s (%i armor, %i shells)" % (self.name, self.armor, self.ammo)

else:

return "%s (DEAD)" % self.name

def fire\_at(self, enemy):

if self.ammo >= 1:

self.ammo -= 1

print self.name, "fires on", enemy.name

enemy.hit()

else:

print self.name, "has no shells!"CHAPTER 2 ■ EXPLORING PYTHON

def hit(self):

self.armor -= 20

print self.name, "is hit!"

if self.armor <= 0:

self.explode()

def explode(self):

self.alive = False

print self.name, "explodes!"

The first method in Listing 2-5 is another special method. Any name with two underscores

at the front and end has a special meaning to Python. The purpose of \_\_str\_\_ is to return a

string that describes the object; it is called when you try to convert the object to a string with

str, which will happen when you print it. So if we were to do print my\_tank, it should display a

string with some useful information about Bob the tank. The \_\_str\_\_ in Listing 2-5 returns a

different string depending on whether the tank is alive or dead. If the tank is alive, then this line

will run:

return "%s (%i armor, %i shells)" % (self.name, self.armor, self.ammo)

This does something you may not have seen before. The string "%s (%i armor, %i

shells)" is combined with a tuple (self.name, self.armor, self.ammo), using the % operator.

This is known as string formatting, which is a great way of creating complex strings without

much fuss. The first two characters in the string are %s, this tells Python to replace them with

the first item in the tuple, which is a string containing the name of the tank. Later in the string

Python reaches %i, which is replaced by the second item in the tuple (an integer) and so on

until there are no more items in the tuple. String interpolation is often simpler to use than add-

ing many small strings together. This line does the same thing, but uses simple string

concatenation:

return self.name+"

("+str(self.armor)+" armor, "+str(self.ammo)+" shells)"

This is a little more complex, as I’m sure you will agree! String formatting can format integers,

floats, and strings in a variety of ways. See the Python documentation for more information

(http://docs.python.org/lib/typesseq-strings.html).

The second method in the Tank class, fire\_at, is where things get interesting. It takes the

parameter enemy, which is the tank object we want to fire at. First it checks how much ammo

is remaining. If there is at least one shell, it reduces self.ammo by 1 (because we just fired a

shell) and calls the enemy tank’s hit method. Inside the enemy tank’s hit method it reduces

self.armor by 20. If there is no armor remaining, then the enemy is dead, so we call its explode

method to mark the tank as deceased.

2930

CHAPTER 2 ■ EXPLORING PYTHON

If this were a graphical game we were working on, these methods would create some visual

effects. fire\_at would create a shell image or 3D model and set its trajectory, and explode

would likely display some kind of impressive explosion animation. But for this small test game

we will just use a few print statements to describe what is currently happening.

Listing 2-6 shows the Tank class in its entirety; save it as tank.py. If you run this script it will

do nothing, because it just defines the Tank class. We will create another Python script with the

rest of the game code.

Listing 2-6. tank.py

class Tank(object):

def \_\_init\_\_(self, name):

self.name = name

self.alive = True

self.ammo = 5

self.armor = 60

def \_\_str\_\_(self):

if self.alive:

return "%s (%i armor, %i shells)" % (self.name, self.armor, self.ammo)

else:

return "%s (DEAD)" % self.name

def fire\_at(self, enemy):

if self.ammo >= 1:

self.ammo -= 1

print self.name, "fires on", enemy.name

enemy.hit()

else:

print self.name, "has no shells!"

def hit(self):

self.armor -= 20

print self.name, "is hit!"

if self.armor <= 0:

self.explode()

def explode(self):

self.alive = False

print self.name, "explodes!"CHAPTER 2 ■ EXPLORING PYTHON

Python in Practice

The game we are going to create is more of a simulation than a game, but it should be enough

to introduce a few important game concepts. We will create a number of tanks and let them

take shots at each other. The winner is simply the last tank left in the game. Listing 2-7 shows

the code that completes the tank game.

Listing 2-7. tankgame.py

from tank import Tank

tanks = { "a":Tank("Alice"), "b":Tank("Bob"), "c":Tank("Carol") }

alive\_tanks = len(tanks)

while alive\_tanks > 1:

print

for tank\_name in sorted( tanks.keys() ):

print tank\_name, tanks[tank\_name]

first = raw\_input("Who fires? ").lower()

second = raw\_input("Who at? " ).lower()

try:

first\_tank = tanks[first]

second\_tank = tanks[second]

except KeyError, name:

print "No such tank!", name

continue

if not first\_tank.alive or not second\_tank.alive:

print "One of those tanks is dead!"

continue

print

print "\*" \* 30

first\_tank.fire\_at(second\_tank)

if not second\_tank.alive:

alive\_tanks -= 1

print "\*" \* 30

for tank in tanks.values():

if tank.alive:

print tank.name, "is the winner!"

break

3132

CHAPTER 2 ■ EXPLORING PYTHON

When you see any piece of code for the first time (in any language), it can be a little intim-

idating. But once you break it down you should find that it consists of familiar things. So let’s

dissect Listing 2-7 like a trained chef preparing fugu!

The first thing tankgame.py needs to do is import our tank module, which contains the Tank

class. When a new script runs, it only has access to the built-in classes, such as strings and lists. If

you want to use another class that isn’t defined directly, you first have to import it from another

Python file. The line from tank import Tank tells Python to look for the module called tank (.py

is assumed) and read in the Tank class. An alternative would be to do a simple import tank, which

would let us access everything inside tank.py.

■Note When you do from tank import Tank, it imports the Tank class (capital T) to the current

namespace—which means you can now use Tank as if you had just cut and pasted it into your script. How-

ever, if you just do import tank, you have imported the tank namespace, which means you would have to

refer to the Tank class as tank.Tank, as in my\_tank = tank.Tank("Bob"). See the section “Introducing

import” later in this chapter for more details on the import statement.

Next we create a dictionary called tanks, which will be used to store all our tank objects.

We will work with three, but feel free to add more tanks if you like.

tanks = { "a":Tank("Alice"), "b":Tank("Bob"), "c":Tank("Carol") }

alive\_tanks = len(tanks)

The three tanks have the strings "a", "b", and "c" as keys, so we can look them up easily. Once

we create our tanks, we store the number of tanks in alive\_tanks so we can keep count of tanks

still in the game:

while alive\_tanks > 1:

This starts off a while loop that keeps going while there is more than one surviving tank. Games

always have a big loop at their core. For a visual game the main loop runs once per frame to

update and display visuals, but here the loop represents a single round in the simulation.

Inside the while loop we first print a blank line to make the text for each round a little easier

to separate. Then we have another loop that displays a little information on each of the tanks:

print

for tank\_name in sorted( tanks.keys() ):

print tank\_name, tanks[tank\_name]

The keys method of dictionaries returns a list of the keys that it contains, but because of the

nature of dictionaries the keys won’t necessarily be in the order that they were added. So when

we get the list of keys for tanks we immediately pass it to sorted, which is a built-in function

that returns a sorted copy of a list.

The print statement inside the for loop looks up the key in the tanks dictionary and prints

the tank object it finds. Remember, printing an object calls its \_\_str\_\_ function to get some

useful information.CHAPTER 2 ■ EXPLORING PYTHON

Next we ask the user for two tanks: the tank that fires (first) and the tank that it hits

(second):

first = raw\_input("Who fires? ").lower()

second = raw\_input("Who at? " ).lower()

The built-in function raw\_input displays a prompt and waits until the user has entered some

text, which it returns as a string. In the preceding code we call the lower method of the returned

string to convert it to lowercase because we need a lowercase string to look up the appropriate

tank, but we don’t mind if the user enters a name using uppercase letters.

With the two tank keys in hand, we can use them to look up the actual tank object. This is

simple enough: we can just do tanks[first] to retrieve the tank:

try:

first\_tank = tanks[first]

second\_tank = tanks[second]

except KeyError, name:

print "No such tank!", name

continue

But because the user could type anything at the prompt, we need some way of handling a situ-

ation where the user makes an error or deliberately tries to break our game!

Whenever Python is unable to do something it is asked to do, it will throw an exception. If

you don’t do anything to handle these exceptions, the Python script will exit—which would be

disastrous in a real game. Fortunately it is possible to predict potential exceptions and handle

them if they occur. If either first or second is not a key in the tanks dictionary, then Python will

throw a KeyError exception when we try to look either of them up. This won’t make the script

exit because we look up the keys inside a try: block, which says to Python that the code block

may throw an exception. If a KeyError does occur, Python jumps to the code under except

KeyError: (which is ignored if no exception occurs).

Inside our KeyError exception handler we first display a brief message to inform the user

that they did something wrong, and then move on to a continue statement, which tells Python

to ignore the rest of the code in this loop and jump back to the top of the innermost loop.

if not first\_tank.alive or not second\_tank.alive:

print "One of those tanks is dead!"

continue

This piece of code handles the situation if one or both of the tanks is dead—since there is no

point in firing on a dead tank, and tanks that are dead can’t fire anyway! It simply displays a

message and does another continue.

If we have managed to get to this point in the code, we have two valid tank objects:

first\_tank and second\_tank:

first\_tank.fire\_at(second\_tank)

if not second\_tank.alive:

alive\_tanks -= 1

3334

CHAPTER 2 ■ EXPLORING PYTHON

The first tank does the firing, so we call its fire\_at method and pass in the second tank as the

enemy. If the second tank is killed by the first (armor reaches 0), its alive property will be set to

False. When this happens, the alive\_tanks count is reduced by 1.

Eventually, after a few rounds of the game the value of alive\_tanks will reach 1. And when

that happens, the main game loop will end, as it only loops when alive\_tanks is greater than 1.

The purpose of the last section of code is to display which tank won the game:

for tank in tanks.values():

if tank.alive:

print tank.name, "is the winner!"

break

It is another loop that goes through each value in tanks.values(), which is the complement to

keys()—it gives us a list of all our tank objects. We know that there is only one tank that has

alive set to True, so we test it with a simple if statement. Once we find that last remaining tank,

we print a little message and then execute the break statement. The break statement is the part-

ner to continue, but rather than jumping to the beginning of the loop, it jumps to the end and

stops looping.

So that’s our little game. Now I’d be the first to admit that it is not the most exciting of

games. It’s no Quake beater, but even Quake will do similar things. All 3D shooters must keep

track of health/armor and ammunition, as well as who is still alive. By the end of this book,

though, our game objects will be rendered in stunning 3D rather than a line of text. The follow-

ing is the output from tankgame.py:

a Alice (60 armor, 5 shells)

b Bob (60 armor, 5 shells)

c Carol (60 armor, 5 shells)

Who fires? a

Who at? b

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Alice fires on Bob

Bob is hit!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

a Alice (60 armor, 4 shells)

b Bob (40 armor, 5 shells)

c Carol (60 armor, 5 shells)

Who fires?CHAPTER 2 ■ EXPLORING PYTHON

Using the Standard Library

Python is packaged with a huge collection of classes and functions known as the standard

library. This is why Python is often described as having batteries included, because you can

take advantage of code written by Python experts to do everything from trigonometry to down-

loading web pages and sending e-mails. Libraries are organized into modules, or packages,

with each having a specific purpose. You make use of these modules by importing them in the

same way that the tank game (Listing 2-7) imported the Tank class.

When you import something in Python, it will first look in the current directory for a cor-

responding Python file. If it doesn’t find it, Python will look for a module in the standard

library.

Let’s go over just a few of the modules in the standard library. We can’t cover them all—

that would take a book in itself—but if you need more information on any module, take a

look at the docs that came with your Python distribution or browse them online at http://

docs.python.org/lib/lib.html.

Introducing import

There are few ways in which you can import things from your own code, or from the standard

library. Which method you use depends on how you want to access the classes and functions

contained in the module. Modules can be imported with the import keyword, followed by the

name of a module. For example, the following line would import a module called mymodule:

import mymodule

Importing a module this way creates a new namespace, which means that you will need to type

the name of the module and a dot before any of the classes or functions that you use. For

instance, if there were a function in mymodule called myfunction, you would call it like this:

mymodule.myfunction()

This is the usual way of importing modules from the standard library, because it keeps things

in each module separate; if you had another function called myfunction in a different module,

there would be no confusion as to which function is being called.

It is also possible to import specific classes or functions from a module, using the from

statement. The following line imports myclass from mymodule to the current namespace:

from mymodule import myclass

Use this method if you just want a few things from the module and you know that their names

won’t conflict with anything else in your script. You can import several classes, functions, and

so forth by adding a comma between each one. So from mymodule import myclass, myfunction

would import two things from the mymodule class.

3536

CHAPTER 2 ■ EXPLORING PYTHON

You can use a \* symbol to indicate that you want to import everything from a module

into the current namespace. For example, the following line would import everything from

mymodule to the current namespace:

from mymodule import \*

This import method saves on typing, because you don’t need the name of the module to refer

to it—but only use this method if the module contains a small number of things and you know

the names won’t conflict with other classes or functions in your script. The math module is a

good candidate for this kind of import.

Useful Modules for Games

The standard library contains a large number of modules, but you will only use a few of them

in game. Let’s go over some of the more commonly used modules.

Math Module

People are often surprised when I tell them I’m not very good at math. “But you are a computer

programmer!” they exclaim. “Exactly,” I tell them. “I get the computer to do the math for me.”

Basic math is built into Python; you can add, subtract, and multiply without importing a spe-

cial module. But you do need the math module for more advanced functions—the kind of thing

you would find on a scientific calculator. See Table 2-5 for a few of them.

Table 2-5. Some Functions in the math Module

FunctionDescriptionExample

sinReturns the sine of a number, in radianssin(angle)

cosReturns the cosine of a number, in radianscos(angle)

tanReturns the tangent of a number, in radianstan(angle)

ceilReturns the largest integer greater than or equal to a numberceil(3.4323)

fabsReturns the absolute value (without the sign) of numberfabs(–2.65)

floorReturns the largest integer less than or equal to a numberfloor(7.234)

piThe value of pipi\*radius\*\*2CHAPTER 2 ■ EXPLORING PYTHON

Let’s use the math module to calculate the area of a circle, given its radius. If you remember

from school, the formula for this is pi times the radius squared, where pi is a magic number that

equals 3.14 something. Fortunately Python has a better memory for numbers than me, and you

can rely on it having a more accurate representation of pi. It’s such a simple function we will

use the interactive interpreter:

>>> from math import \*

>>> def area\_of\_circle(radius):

...

return pi\*radius\*\*2

...

>>> area\_of\_circle(5)

78.539816339744831

Because the math module has just a few small functions, we are lazy and import everything to

the current module, just to use pi. We then define a very trivial function that takes the radius

and returns the area of the circle. To test it, we calculate the area of a circle with a radius of 5

units, which turns out to be a little over 78.5 units squared.

Datetime Module

The datetime module has a number of functions and classes that deal with date and time. You

can use it to query the time of your PC’s internal clock and to calculate time differences between

dates. It may sound like a simple task, because we often do mental calculations about dates, but

it can get a little complicated when you think about leap years and time zones! Fortunately we

can rely on the work of some smart programmers and have Python do this effortlessly. In the

datetime module is a class of the same name. Let’s use it to find the current time:

>>> from datetime import datetime

>>> the\_time = datetime.now()

>>> the\_time.ctime()

'Thu Dec 28 15:35:26 2006'

After importing the datetime class from the datetime module, we call the function now to return

a datetime object with the current time. The function now is what is called a static method

because you use it on a class rather than an object created with that class. Once we have the

current date and time stored in the\_time, we call the ctime method, which returns a friendly

representation of the time as a string. Obviously it will return a different result when you run it.

So what use is finding the time in a game? Well, you may want to store a time stamp with

saved games and high scores. You could also link the game with the current time of day, so it is

bright and sunny at midday but dark and gloomy if you play it in the evening. Have a look at

Table 2-6 for some of the things you can find in the datetime module.

3738

CHAPTER 2 ■ EXPLORING PYTHON

Table 2-6. Some Classes in the datetime Module

ClassDescription

timedeltaStores a difference between two times

dateStores a date value

datetimeStores date and time values

timeStores a time value

Random Module

You won’t be surprised to learn that the random module is used to generate random numbers,

although you may be surprised to learn that the numbers it generates aren’t truly random.

That’s because computers aren’t actually capable of selecting something at random; they will

do the same thing again given identical conditions. The numbers that random generates are

pseudorandom, which means they are pulled from a very long sequence of numbers that

appear random but will eventually repeat if you generate enough of them. Fortunately you can

use them in a game because nobody will notice if they repeat after a few billion times!

Random (or pseudorandom) numbers are very useful in games to stop them from getting

predictable. If a game has no random elements, players will eventually memorize all the

sequences of actions, making it less fun (for most people).

Let’s write a short script to simulate ten throws of a standard six-sided die (Listing 2-8).

Listing 2-8. Dice Simulator

import random

for roll in xrange(10):

print random.randint(1,6)

Wow, just three lines. All this does is import the random module, then call random.randint

ten times and prints the results. The function randint takes two parameters, a and b, and

returns a pseudorandom number in the range of a to b (possibly including the end values). So

randint(1, 6) returns 1, 2, 3, 4, 5, or 6—as a die would.

■Note You may have noticed in Listing 2-8 that the value of roll is never actually used inside the loop.

The call to xrange(10) generates numbers from 0 to 9, but we ignore them because all we are interested in

is repeating the loop ten times. Rather than thinking of a name for a value that is never used, it is common to

use a single underscore in its place. So the loop in Listing 2-8 may be rewritten as for \_ in xrange(10):.CHAPTER 2 ■ EXPLORING PYTHON

Although the numbers that Listing 2-8 produces appear random, they are actually pseudo-

random, which means they are chosen from a large mathematically generated sequence.

Occasionally you may need to repeat a sequence of pseudorandom numbers—when playing

back a demo, for instance. You can tell the random module to start generating numbers from a

particular point in the sequence by calling the random.seed function. If you call it twice with the

same value, it will cause the random module to reproduce the same sequence of numbers.

Listing 2-9 demonstrates how the seed function can be used to create predictable sequences.

Listing 2-9. A Better Dice Simulator

import random

random.seed(100)

for roll in xrange(10):

print random.randint(1, 6)

print "Re-seeded"

random.seed(100)

for roll in xrange(10):

print random.randint(1, 6)

If you run this small script, you will see the same sequence of numbers, repeated twice.

Have a look at Table 2-7 for a few of the capabilities of the random module.

Table 2-7. Some Functions in the random Module

FunctionDescription

seedSeeds the random number generator

randintReturns a random integer between two values

choiceSelects a random element from a collection

randomReturn a float between 0 and 1

Summary

We’ve seen that you can use boolean logic to make decisions in Python code. The if statement

takes a boolean expression, such as a > 3, and runs a code block only if that condition results

in True. You can append an if statement with one or more else statements, which run their

code block only if the condition is False. Logic can be combined using the and and or opera-

tors, and can be reversed using the not operator.

Functions are stored Python code, created with the def statement. When you define func-

tions, you specify a list of parameters, which is a list of the information that the function needs

to run and optionally return a value. A number of built-in functions are available for you to use.

3940

CHAPTER 2 ■ EXPLORING PYTHON

Object-oriented programming is a fancy term for a simple concept. It simply means stor-

ing the information needed to describe something together with a number of actions that work

with that information. In a game, just about everything will be defined as an object. Python

classes are defined with the class statement, which you can think of as a template for creating

new objects. Functions created within a class statement are called methods, which are similar

to other functions with the exception that the very first parameter is the object that the method

applies to. The \_\_init\_\_ function is a special method that is called when an object is first cre-

ated; you use it to initialize the information, or properties, contained in the object.

Python has a large standard library that can do a variety of useful things. The library is

organized into a number of modules, which can contain classes, functions, or other Python

objects.

In the next chapter, we cover how to use the Pygame module to open a window and dis-

play graphics.CHAPTER 3

■■■

Introducing Pygame

H

ave you ever opened up your computer and had a look inside the case? No need to do it

now, but you will find that it is built from a number of parts necessary to deliver your comput-

ing experience. The video card generates an image and sends a signal to your monitor. The

sound card mixes sound together and sends audio to your speakers. Then there are the input

devices, such as the keyboard, mouse, and joystick(s), and a variety of other electronic giz-

mos—all of which are essential in making a game.

In the early days of home computers, programmers with bad haircuts and thick-rimmed

glasses had to come to grips with each of the computer’s components. The game programmer

had to read the technical manual for each device in order to write the computer code to commu-

nicate with it—all before working on the actual game. The situation only got worse when the

manufacturers brought out different devices and versions of existing devices with new capabili-

ties. Programmers wanted to support as many devices as possible so there was a bigger market

for their games, but they found themselves bogged down in the details of working with these new

graphics and sound cards. It was also a pain for the game-buying public, who had to carefully

check the box to see if they had the right combination of devices to make the game work.

Things got a little easier with the introduction of graphical operating systems like Micro-

soft Windows. They gave the game programmer a single way of communicating with the

devices. It meant that the programmer could throw away the technical manuals because the

manufacturers supplied drivers, small programs that handle the communication between the

operating system and the hardware.

Fast-forward to more recent times, when programmers still hav