PART II

GRAPHICAL USER INTERFACES WITH PYGAME

These chapters introduce you to *pygame,* an external package that adds functionality common to GUI programs. Pygame allows you to write Python programs that have windows, respond to the mouse and keyboard, play sounds, and more.

Chapter 5 gives you a basic understanding of how pygame works and provides a standard template for building pygame-based programs. We'll build a few simple programs first, create a program that controls an image with the keyboard, then we'll build a ball-bouncing program.

Chapter 6 explains how pygame can best be used as an object-oriented framework. You'll see how to rewrite the ball-bouncing program using object-oriented techniques, and develop simple buttons and text input fields.

Chapter 7 describes the pygwidgets module, which contains full implementations of many standard user interface widgets like buttons, input and output fields, radio buttons, checkboxes, and more, all using object-oriented programming. All the code is available for you so that you can use it to build your own applications. I'll provide several examples.

5

INTRODUCTION TO PYGAME

The Python language was designed to handle text input and text output. It provides the ability to get text from and send text to the user, a file, and the internet. The core language, however, has no way of dealing with more modern concepts such as windows, mouse clicks, sounds, and so on. So, what if you want to use Python to create something more state-of-the-art than a text-based program? In this chapter I'll introduce *pygame*, a free open source external package that was designed to extend Python to allow programmers to build game programs. You can also use pygame to build other kinds of interactive programs with a graphical user interface (GUI). It adds the ability to create windows, show images, recognize mouse movements and clicks, play sounds, and more. In short, it allows Python programmers to build the types of games and applications that current computer users have become familiar with.

It is not my intent to turn you all into game programmers—even though that might be a fun outcome. Rather, I'll use the pygame environment to

make certain object-oriented programming techniques clearer and more visual. By working with pygame to make objects visible in a window and dealing with a user interacting with those objects, you should gain a deeper understanding of how to effectively use OOP techniques.

This chapter provides a general introduction to pygame, so most of the information and examples in this chapter will use procedural coding. Starting with the next chapter, I will explain how to use OOP effectively with pygame.

Installing Pygame

Pygame is a free downloadable package. We'll use the package manager *pip* (short for *pip installs packages*) to install Python packages. As mentioned in the Introduction, I am assuming that you have installed the official version of Python from *python.org*. The pip program is included as part of that download, so you should already have it installed.

Unlike a standard application, you must run pip from the command line. On a Mac, start the Terminal application (located in the *Utilities* subfolder inside the *Applications* folder). On a Windows system, click the Windows icon, type **cmd**, and press ENTER.

NOTE *This book was not tested with Linux systems. However, most, if not all, of the content should work with minimal tweaking. To install pygame on a Linux distribution, open a terminal in whatever way you're used to.*

Enter the following commands at the command line:

```
python3 -m pip install -U pip --user
python3 -m pip install -U pygame --user
```

The first command ensures that you have the latest version of the pip program. The second line installs the most recent version of pygame.

If you have any problems installing pygame, consult the pygame documentation at *https://www.pygame.org/wiki/GettingStarted*. To test that pygame has been installed correctly, open IDLE (the development environment that is bundled with the default implementation of Python), and in the shell window enter:

```
import pygame
```

If you see a message saying something like "Hello from the pygame community" or if you get no message at all, then pygame has been installed correctly. The lack of an error message indicates that Python has been able to find and load the pygame package and it's ready to use. If you would like to see a sample game using pygame, enter the following command (which starts a version of *Space Invaders*):

```
python3 -m pygame.examples.aliens
```

Before we get into using pygame, I need to explain two important concepts. First, I'll explain how individual pixels are addressed in programs that use a GUI. Then, I'll discuss event-driven programs and how they differ from typical text-based programs. After that, we'll code a few programs that demonstrate key pygame features.

Window Details

A computer screen is made up of a large number of rows and columns of small dots called *pixels* (from the words *picture element*). A user interacts with a GUI program through one or more windows; each window is a rectangular portion of the screen. Programs can control the color of any individual pixel in their window(s). If you're running multiple GUI programs, each program is typically displayed in its own window. In this section, I'll discuss how you address and alter individual pixels in a window. These concepts are independent of Python; they are common to all computers and are used in all programming languages.

The Window Coordinate System

You are probably familiar with Cartesian coordinates in a grid like Figure 5-1.

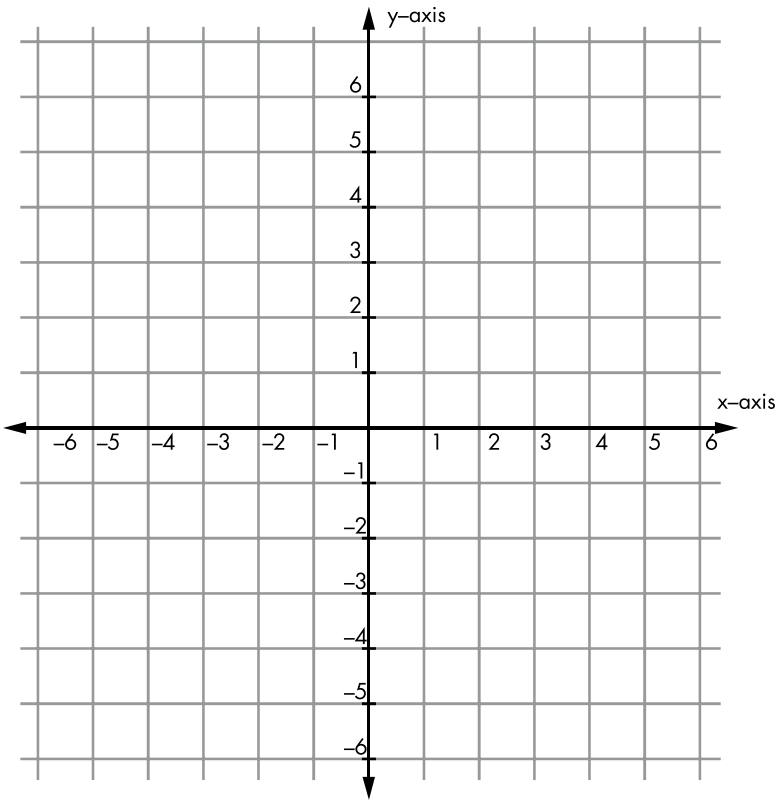

Figure 5-1: The standard Cartesian coordinate system

Any point in a Cartesian grid can be located by specifying its x- and y-coordinates (in that order). The origin is the point specified as (0, 0) and is found in the center of the grid.

Computer window coordinates work in a similar way (Figure 5-2).

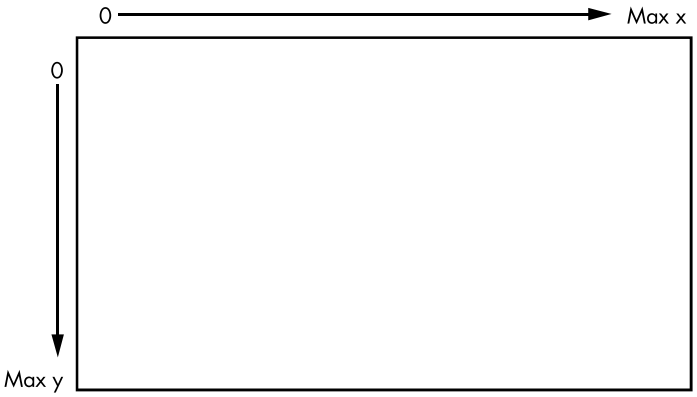

Figure 5-2: A computer window's coordinate system

However, there are a few key differences:

1. The origin (0, 0) point is in the upper-left corner of the window.
2. The y-axis is reversed so that y values start at zero at the top of the window and increase as you go down.
3. The x and y values are always integers. Each (x, y) pair specifies a single pixel in the window. These values are always specified as relative to the upper-left corner of the window, not the screen. That way, the user can move the window anywhere on the screen without affecting the coordinates of the elements of the program displayed in the window.

The full computer screen has its own set of (x, y) coordinates for every pixel and uses the same type of coordinate system, but programs rarely, if ever, need to deal with screen coordinates.

When we write a pygame application, we need to specify the width and height of the window we want to create. Within the window, we can address any pixel using its x- and y-coordinates, as shown in Figure 5-3.

Figure 5-3 shows a black pixel at position (3, 5). That is an x-value of 3 (note that this is actually the fourth column, since coordinates start at 0) and a y value of 5 (actually the sixth row). Each pixel in a window is commonly referred to as a *point*. To reference a point in a window, you would typically use a Python tuple. For example, you might have an assignment statement like this, with the x value first:

```
pixelLocation = (3, 5)
```

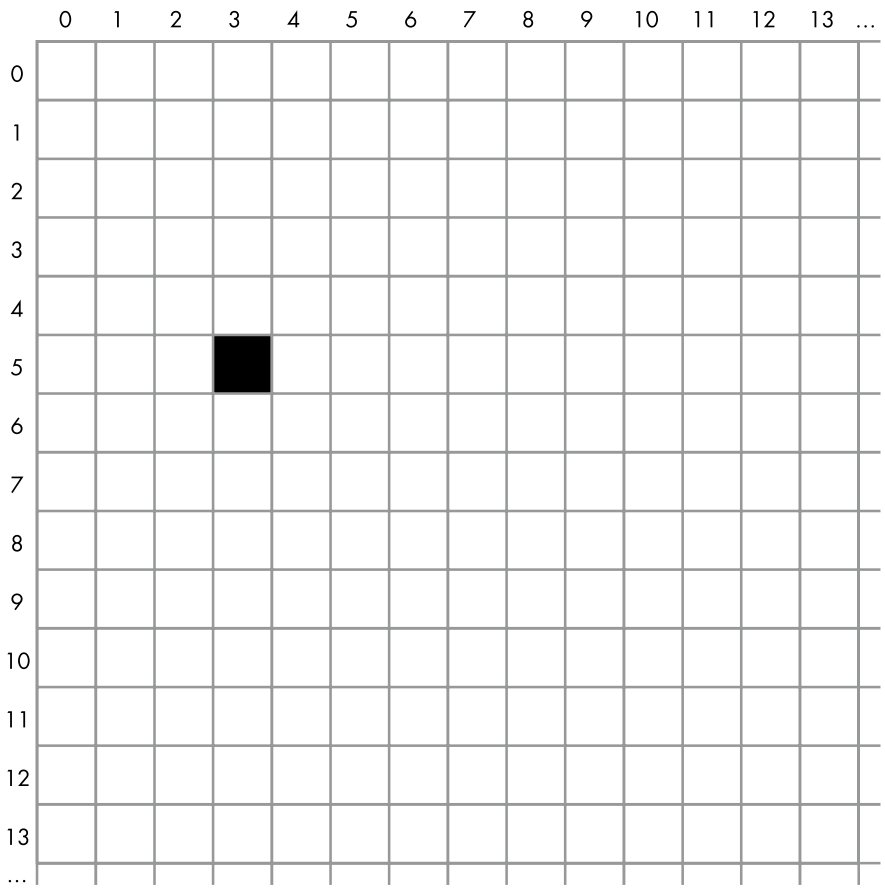

Figure 5-3: A single point (a single pixel) in a computer window

To show an image in a window, we need to specify the coordinates of its starting point—always the upper-left corner of the image—as an (x, y) pair, as in Figure 5-4, where we draw the image at location (3, 5).

When working with an image, you'll often need to deal with the *bounding rectangle*, which is the smallest rectangle that can be made that completely surrounds all pixels of the image. A rectangle is represented in pygame by a set of four values: x, y, width, height. The rectangle for the image in Figure 5-4 has values of 3, 5, 11, 7. I'll show you how to use a rectangle like this in an upcoming example program. Even if your image is not rectangular (for example, if it's a circle or an ellipse), you still have to consider its bounding rectangle for positioning and collision detection.

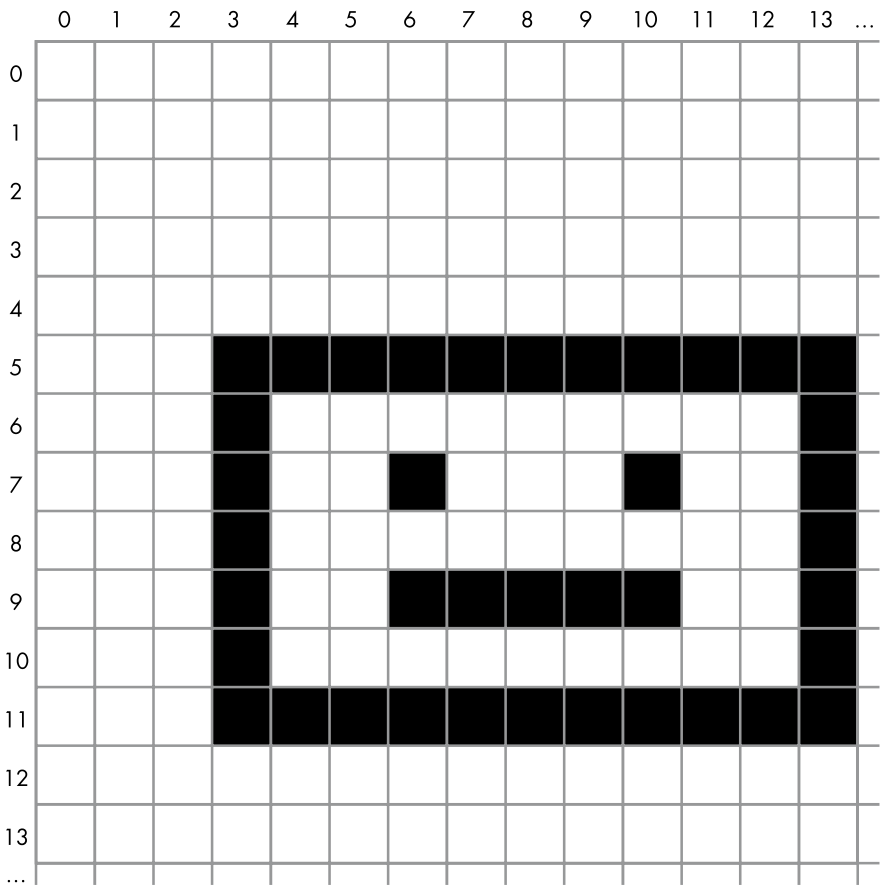

Figure 5-4: An image in a window

Pixel Colors

Let's explore how colors are represented on the computer screen. If you have experience with a graphics program like Photoshop, you probably already know how this works, but you may want a quick refresher anyway.

Each pixel on the screen is made up of a combination of three colors: red, green, and blue, often referred to as *RGB*. The color displayed in any pixel is composed of some amount of red, green, and blue, where the amount of each is specified as a value from 0, meaning none, to 255, meaning full intensity. Therefore, there are 256 × 256 × 256 possible combinations, or 16,777,216 (often referred to as just "16 million") possible colors, for each pixel.

Colors in pygame are given as RGB values, and we write them as Python tuples of three numbers. Here is how we create constants for the main colors:

```
RED = (255, 0, 0)   # full red, no green, no blue
GREEN = (0, 255, 0)  # no red, full green, no blue
BLUE = (0, 0, 255)  # no red, no green, full blue
```

Here are the definitions of a few more colors. You can create a color using any combination of three numbers between 0 and 255:

```
BLACK = (0, 0, 0)    # no red, no green, no blue
WHITE = (255, 255, 255)  # full red, full green, full blue
DARK_GRAY = (75, 75, 75)
MEDIUM_GRAY = (128, 128, 128)
LIGHT_GRAY = (175, 175, 175)
TEAL = (0, 128, 128)  # no red, half-strength green, half-strength blue
YELLOW = (255, 255, 0)
PURPLE = (128, 0, 128)
```

In pygame, you'll need to specify colors when you want to fill the background of a window, draw a shape in a color, draw text in a color, and so on. Defining colors up front as tuple constants makes them very easy to spot later in code.

Event-Driven Programs

In most of the programs in the book so far, the main code has lived in a `while` loop. The program stops at a call to the built-in `input()` function and waits for some user input to work on. Program output is typically handled using calls to `print()`.

In interactive GUI programs, this model no longer works. GUIs introduce a new model of computing known as the *event-driven* model. Event-driven programs don't rely on `input()` and `print()`; instead, the user interacts with elements in a window at will using a keyboard and/or mouse or other pointing device. They may be able to click various buttons or icons, make selections from menus, provide input in text fields, or give commands via clicks or key presses to control some avatar in the window.

NOTE *Calls to `print()` can still be highly useful for debugging, when used to write out intermediate results.*

Central to event-driven programming is the concept of an *event*. Events are difficult to define and are best described with examples, such as a mouse click and a key press (each of which is actually made up of two events: mouse down and mouse up and key down and key up, respectively). Here is my working definition.

event Something that happens while your program is running that your program wants to or needs to respond to. Most events are generated by user actions.

An event-driven GUI program runs constantly in an infinite loop. Each time through the loop, the program checks for any new events it needs to react to and executes appropriate code to handle those events. Also, each time through the loop, the program needs to redraw all the elements in the window to update what the user sees.

For example, say we have a simple GUI program that displays two buttons: Bark and Meow. When clicked, the Bark button plays a sound of a dog barking and the Meow button plays a sound of a cat meowing (Figure 5-5).

Figure 5-5: A simple program with two buttons

The user can click these buttons in any order and at any time. To handle the user's actions, the program runs in a loop and constantly checks to see if either button has been clicked. When it receives a mouse down event on a button, the program remembers that the button has been clicked and draws the depressed image of that button. When it receives a mouse up event on the button, it remembers the new state and redraws the button with its original appearance, and it plays the appropriate sound. Because the main loop runs so quickly, the user perceives that the sound plays immediately after they click the button. Each time through the loop, the program redraws both buttons with an image matching each button's current state.

Using Pygame

At first, pygame may seem like an overwhelmingly large package with many different calls available. Although it is large, there's actually not a lot that you need to understand to get a small program up and running. To introduce pygame, I'll first give you a template that you can use for all pygame programs you create. Then I'll build on that template, adding key pieces of functionality little by little.

In the following sections, I'll show you how to:

- Bring up a blank window.
- Show an image.
- Detect a mouse click.
- Detect both single and continuous key presses.
- Create a simple animation.
- Play sound effects and background sounds.
- Draw shapes.

In the next chapter, we'll continue the discussion of pygame and you'll see how to:

- Animate many objects.
- Build and react to a button.
- Create a text display field.

Bringing Up a Blank Window

As I said earlier, pygame programs run constantly in a loop, checking for events. It might help to think of your program as an animation, where each pass through the main loop is one frame. The user may click on something during any frame, and your program must not only respond to that input but also keep track of everything it needs to draw in the window. For instance, in one example program later in this chapter, we'll move a ball across the window so in each frame the ball is drawn in a slightly different position.

Listing 5-1 is a generic template that you can use as a starting point for all your pygame programs. This program opens a window and paints the entire contents black. The only thing the user can do is click the close button to quit the program.

File: PygameDemo0_WindowOnly/PygameWindowOnly.py

```
# pygame demo 0 - window only

# 1 - Import packages
import pygame
from pygame.locals import *
import sys

# 2 - Define constants
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30

# 3 - Initialize the world
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

# 4 - Load assets: image(s), sound(s), etc.

# 5 - Initialize variables

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        # Clicked the close button? Quit pygame and end the program
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # 8 - Do any "per frame" actions

    # 9 - Clear the window
```

```
    window.fill(BLACK)

    # 10 - Draw all window elements

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)
```

Listing 5-1: A template for creating pygame programs

Let's walk through the different parts of this template:

1. Import packages.

   The template starts with the `import` statements. We first import the pygame package itself, then some constants defined inside pygame that we'll use later. The last import is the `sys` package, which we'll use to quit our program.

2. Define constants.

   We next define any constants for our program. First we define the RGB value for `BLACK`, which we will use to paint the background of our window. Then we define constants for the width and height of our window in pixels and a constant for the refresh rate for our program. This number defines the maximum number of times the program will loop (and therefore redraw the window) per second. Our value of 30 is fairly typical. If the amount of work done in our main loop is excessive, the program might run slower than this value, but it will never run faster. A refresh rate that's too high might cause the program to run too fast. In our ball example, this means the ball might bounce around the window faster than intended.

3. Initialize the pygame environment.

   In this section, we call a function that tells pygame to initialize itself. We then ask pygame to create a window for our program with the `pygame.display.set_mode()` function and pass in the desired width and height of the window. Finally, we call another pygame function to create a clock object, which will be used at the bottom of our main loop to maintain our maximum frame rate.

4. Load assets: image(s), sound(s), and so on.

   This is a placeholder section, into which we will eventually add code to load external images, sounds, and so on from the disk for use in our program. In this basic program we're not using any external assets, so this section is empty for now.

5. Initialize variables.

   Here we will eventually initialize any variables that our program will use. Currently we have none, so we have no code here.

6. Loop forever.

   Here we start our main loop. This is a simple `while True` infinite loop. Again, you can think of each iteration through the main loop as one frame in an animation.

7. Check for and handle events; commonly referred to as the *event loop*.

   In this section, we call `pygame.event.get()` to get a list of the events that happened since the last time we checked (the last time the main loop ran), then iterate through the list of events. Each event reported to the program is an object, and every event object has a type. If no event has happened, this section is skipped over.

   In this minimal program, where the only action a user can take is to close the window, the only event type we check for is the constant `pygame.QUIT`, generated by pygame when the user clicks the close button. If we find this event, we tell pygame to quit, which frees up any resources it was using. Then we quit our program.

8. Do any "per frame" actions.

   In this section we'll eventually put any code that needs to run in every frame. This might involve moving things in the window or checking for collisions between elements. In this minimal program, we have nothing to do here.

9. Clear the window.

   On each iteration through the main loop, our program must redraw everything in the window, which means we need to clear it first. The simplest approach is to just fill the window with a color, which we do here with a call to `window.fill()`, specifying a black background. We could also draw a background picture, but we'll hold off on that for now.

10. Draw all window elements.

    Here we'll place code to draw everything we want to show in our window. In this sample program there is nothing to draw.

    In real programs, things are drawn in the order they appear in the code, in layers from backmost to frontmost. For example, assume we want to draw two partially overlapping circles, A and B. If we draw A first, A will appear behind B, and portions of A will be obscured by B. If we draw B first and then A, the opposite happens, and we see A in front of B. This is a natural mapping equivalent to the layers in graphics programs such as Photoshop.

11. Update the window.

    This line tells pygame to take all the drawing we've included and show it in the window. Pygame actually does all the drawing in steps 8, 9, and 10 in an off-screen buffer. When you tell pygame to update, it takes the contents of this off-screen buffer and puts them in the real window.

12. Slow things down a bit.

    Computers are very fast, and if the loop continued to the next iteration right away without pausing, the program might run faster than the designated frame rate. The line in this section tells pygame to wait until a given amount of time has elapsed in order to make the frames of our program run at the frame rate that we specified. This is important to ensure the program runs at a consistent rate, independent of the speed of the computer on which it's running.

When you run this program, the program just puts up a blank window filled with black. To end the program, click on the close button in the title bar.

Drawing an Image

Let's draw something in the window. There are two parts to showing a graphic image: first we load the image into the computer's memory, then we display the image in the application window.

With pygame, all images (and sounds) need to be kept in files external to your code. Pygame supports many standard graphic file formats, including *.png*, *.jpg*, and *.gif*. In this program we'll load a picture of a ball from the file *ball.png*. As a reminder, the code and assets associated with all the major listings in this book are available for download at *https://www.nostarch.com/objectorientedpython/* and *https://github.com/IrvKalb/Object-Oriented-Python-Code/*.

While we only need one graphic file in this program, it's a good idea to use a consistent approach to handling graphic and sound files, so I'll lay one out for you here. First, create a project folder. Place your main program in that folder, along with any related files containing Python classes and functions. Then, inside the project folder, create an *images* folder into which you'll place any image files you want to use in your program. Also create a *sounds* folder and place any sound files you want to use there. Figure 5-6 shows the suggested structure. All of the example programs in this book will use this project folder layout.

Figure 5-6: Suggested project folder hierarchy

A *path* (also called a *pathname*) is a string that uniquely identifies the location of a file or folder on a computer. To load a graphic or sound file into your program, you must specify the path to the file. There are two types of paths: relative and absolute.

A *relative path* is a relative to the current folder, often called the *current working directory*. When you run a program using an IDE such as IDLE or

PyCharm, it sets the current folder to the one containing your main Python program so you can use relative paths with ease. In this book, I will assume you're using an IDE and will represent all paths as relative paths.

The relative path for a graphic file (for example, *ball.png*) in the same folder as your main Python file would be just the filename as a string (for example, `'ball.png'`). Using the suggested project structure, the relative path would be `'images/ball.png'`.

This says that inside the project folder will be another folder named *images*, and inside that folder is a file named *ball.png*. In path strings, folder names are separated by the slash character.

However, if you expect to run your program from the command line, then you need to construct absolute paths for all files. An *absolute path* is one that starts from the root of the filesystem and includes the full hierarchy of folders to your file. To build an absolute path to any file, you can use code like this, which builds an absolute path string to the *ball.png* file in the *images* folder inside the project folder:

```
from pathlib import Path

# Place this in section #2, defining a constant
BASE_PATH = Path(__file__).resolve().parent

# Build a path to the file in the images folder
pathToBall = BASE_PATH + 'images/ball.png'
```

Now we'll create the code of the ball program, starting with the earlier 12-step template and adding just two new lines of code, as shown in Listing 5-2.

File: PygameDemo1_OneImage/PygameOneImage.py

```
# pygame demo 1 - draw one image

--- snip ---
# 3 - Initialize the world
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

# 4 - Load assets: image(s), sound(s), etc.
❶ ballImage = pygame.image.load('images/ball.png')

# 5 - Initialize variables

--- snip ---

    # 10 - Draw all window elements
    # draw ball at position 100 across (x) and 200 down (y)
  ❷ window.blit(ballImage, (100, 200))

    # 11 - Update the window
```

```
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)  # make pygame wait
```

Listing 5-2: Load one image and draw it in every frame.

First, we tell pygame to find the file containing the image of the ball and load that image into memory ❶. The variable `ballImage` now refers to the image of the ball. Notice that this assignment statement is only executed once, before the main loop starts.

NOTE *In the official documentation of pygame, every image, including the application window, is known as a* surface. *I'll use more specific terms: I will refer to the application window simply as a* window *and to any picture loaded from an external file as an* image. *I reserve the term* surface *for any picture drawn on the fly.*

We then tell the program to draw the ball ❷ every time we go through the main loop. We specify the location representing the position to place the upper-left corner of the image's bounding rectangle, typically as a tuple of x- and y-coordinates.

The function name `blit()` is a very old reference to the words *bit block transfer*, but in this context it really just means "draw." Since the program loaded the ball image earlier, pygame knows how big the image is, so we just need to tell it where to draw the ball. In Listing 5-2, we give an x value of 100 and a y value of 200.

When you run the program, on each iteration through the loop (30 times per second) every pixel in the window is set to black, then the ball is drawn over the background. From the user's point of view, it looks like nothing is happening—the ball just stays in one spot with the upper-left corner of its bounding rectangle at location (100, 200).

Detecting a Mouse Click

Next, we'll allow our program to detect and react to a mouse click. The user will be able to click on the ball to make it appear somewhere else in the window. When the program detects a mouse click on the ball, it randomly picks new coordinates and draws the ball at that new location. Instead of using hardcoded coordinates of (100, 200), we'll create two variables, `ballX` and `ballY`, and refer to the coordinates of the ball in the window as the tuple `(ballX, ballY)`. Listing 5-3 provides the code.

File: PygameDemo2_ImageClickAndMove/PygameImageClickAndMove.py

```
# pygame demo 2 - one image, click and move

# 1 - Import packages
import pygame
from pygame.locals import *
import sys
```

```
❶ import random

  # 2 - Define constants
  BLACK = (0, 0, 0)
  WINDOW_WIDTH = 640
  WINDOW_HEIGHT = 480
  FRAMES_PER_SECOND = 30
❷ BALL_WIDTH_HEIGHT = 100
  MAX_WIDTH = WINDOW_WIDTH - BALL_WIDTH_HEIGHT
  MAX_HEIGHT = WINDOW_HEIGHT - BALL_WIDTH_HEIGHT

  # 3 - Initialize the world
  pygame.init()
  window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
  clock = pygame.time.Clock()

  # 4 - Load assets: image(s), sound(s), etc.
  ballImage = pygame.image.load('images/ball.png')

  # 5 - Initialize variables
❸ ballX = random.randrange(MAX_WIDTH)
  ballY = random.randrange(MAX_HEIGHT)
❹ ballRect = pygame.Rect(ballX, ballY, BALL_WIDTH_HEIGHT, BALL_WIDTH_HEIGHT)

  # 6 - Loop forever
  while True:

      # 7 - Check for and handle events
      for event in pygame.event.get():
          # Clicked the close button? Quit pygame and end the program
          if event.type == pygame.QUIT:
              pygame.quit()
              sys.exit()

          # See if user clicked
        ❺ if event.type == pygame.MOUSEBUTTONUP:
              # mouseX, mouseY = event.pos  # Could do this if we needed it

              # Check if the click was in the rect of the ball
              # If so, choose a random new location
            ❻ if ballRect.collidepoint(event.pos):
                  ballX = random.randrange(MAX_WIDTH)
                  ballY = random.randrange(MAX_HEIGHT)
                  ballRect = pygame.Rect(ballX, ballY, BALL_WIDTH_HEIGHT,
                                         BALL_WIDTH_HEIGHT)

      # 8  Do any "per frame" actions

      # 9 - Clear the window
      window.fill(BLACK)

      # 10 - Draw all window elements
      # Draw the ball at the randomized location
    ❼ window.blit(ballImage, (ballX, ballY))
```

```
    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)  # make pygame wait
```

Listing 5-3: Detecting a mouse click and acting on it

Since we need to generate random numbers for the ball coordinates, we import the `random` package ❶.

We then add a new constant to define the height and width of our image as 100 pixels ❷. We also create two more constants to limit the maximum width and height coordinates. By using these constants rather than the size of the window, we ensure that our ball image will always appear fully within the window (remember that when we refer to the location of an image, we are specifying the position of its upper-left corner). We use those constants to choose random values for the starting x- and y-coordinates for our ball ❸.

Next, we call `pygame.Rect()` to create a rectangle ❹. Defining a rectangle requires four parameters—an x-coordinate, a y-coordinate, a width, and a height, in that order:

```
<rectObject> = pygame.Rect(<x>, <y>, <width>, <height>)
```

This returns a pygame rectangle object, or `rect`. We'll use the rectangle of the ball in the processing of events.

We also add code to check if the user clicked the mouse. As mentioned, a mouse click is actually made up of two different events: a mouse down event and a mouse up event. Since the mouse up event is typically used to signal activation, we'll only look for that event here. This event is signaled by a new `event.type` value of `pygame.MOUSEBUTTONUP` ❺. When we find that a mouse up event has occurred, we'll then check to see if the location where the user clicked was inside the current rectangle of the ball.

When pygame detects that an event has happened, it builds an event object containing a lot of data. In this case, we only care about the x- and y-coordinates where the event happened. We retrieve the (x, y) position of the click using `event.pos`, which provides a tuple of two values.

NOTE *If we need to separate the x- and y-coordinates of the click, we can unpack the tuple and store the values into two variables like this:*

```
mouseX, mouseY = event.pos
```

Now we check to see if the event happened inside the rectangle of the ball using `collidepoint()` ❻, whose syntax is:

```
<booleanVariable> = <someRectangle>.collidepoint(<someXYLocation>)
```

The method returns a Boolean `True` if the given point is inside the rectangle. If the user has clicked the ball, we randomly select new values for `ballX` and `ballY`. We use those values to create a new rectangle for the ball at the new random location.

The only change here is that we always draw the ball at the location given by the tuple `(ballX, ballY)` ❼. The effect is that whenever the user clicks inside the rectangle of the ball, the ball appears to move to some new random spot in the window.

Handling the Keyboard

The next step is to allow the user to control some aspect of the program through the keyboard. There are two different ways to handle user keyboard interactions: as individual key presses, and when a user holds down a key to indicate that an action should happen for as long as that key is down (known as *continuous mode*).

Recognizing Individual Key Presses

Like the mouse clicks, each key press generates two events: key down and key up. The two events have different event types: `pygame.KEYDOWN` and `pygame.KEYUP`.

Listing 5-4 shows a small sample program that allows the user to move the ball image in the window using the keyboard. The program also shows a target rectangle in the window. The user's goal is to move the ball image so that it overlaps with the target image.

File: PygameDemo3_MoveByKeyboard/PygameMoveByKeyboardOncePerKey.py

```
# pygame demo 3(a) - one image, move by keyboard

# 1 - Import packages
import pygame
from pygame.locals import *
import sys
import random

# 2 - Define constants
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
BALL_WIDTH_HEIGHT = 100
MAX_WIDTH = WINDOW_WIDTH - BALL_WIDTH_HEIGHT
MAX_HEIGHT = WINDOW_HEIGHT - BALL_WIDTH_HEIGHT
❶ TARGET_X = 400
TARGET_Y = 320
TARGET_WIDTH_HEIGHT = 120
N_PIXELS_TO_MOVE = 3

# 3 - Initialize the world
pygame.init()
```

```
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

# 4 - Load assets: image(s), sound(s), etc.
ballImage = pygame.image.load('images/ball.png')
❷ targetImage = pygame.image.load('images/target.jpg')

# 5 - Initialize variables
ballX = random.randrange(MAX_WIDTH)
ballY = random.randrange(MAX_HEIGHT)
targetRect = pygame.Rect(TARGET_X, TARGET_Y, TARGET_WIDTH_HEIGHT, TARGET_
WIDTH_HEIGHT)

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        # Clicked the close button? Quit pygame and end the program
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        # See if the user pressed a key
      ❸ elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                ballX = ballX - N_PIXELS_TO_MOVE
            elif event.key == pygame.K_RIGHT:
                ballX = ballX + N_PIXELS_TO_MOVE
            elif event.key == pygame.K_UP:
                ballY = ballY - N_PIXELS_TO_MOVE
            elif event.key == pygame.K_DOWN:
                ballY = ballY + N_PIXELS_TO_MOVE

    # 8  Do any "per frame" actions
    # Check if the ball is colliding with the target
  ❹ ballRect = pygame.Rect(ballX, ballY,
                           BALL_WIDTH_HEIGHT, BALL_WIDTH_HEIGHT)
  ❺ if ballRect.colliderect(targetRect):
        print('Ball is touching the target')

    # 9 - Clear the window
    window.fill(BLACK)

    # 10 - Draw all window elements
  ❻ window.blit(targetImage, (TARGET_X, TARGET_Y))  # draw the target
    window.blit(ballImage, (ballX, ballY))  # draw the ball

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)  # make pygame wait
```

Listing 5-4: Detecting and acting on single key presses

First we add a few new constants ❶ to define the x- and y-coordinates of the upper-left corner of the target rectangle and the width and height of the target. We then load the image of the target rectangle ❷.

In the loop where we look for for events, we add a test for a key press by checking for an event of type `pygame.KEYDOWN` ❸. If a key down event is detected, we look into the event to find out what key was pressed. Each key has an associated constant in pygame, so here we check if the user has pressed the left, up, down, or right arrow. For each of these keys, we modify the value of the ball's x- or y-coordinate appropriately by a small number of pixels.

Next we create a pygame `rect` object for the ball based on its x- and y-coordinates and its height and width ❹. We can check to see if two rectangles overlap with the following call:

```
<booleanVariable> = <rect1>.colliderect(<rect2>)
```

This call compares two rectangles and returns `True` if they overlap at all or `False` if they don't. We compare the ball rectangle with the target rectangle ❺, and if they overlap, the program prints "Ball is touching the target" to the shell window.

The last change is where we draw both the target and the ball. The target is drawn first so that when the two overlap, the ball appears over the target ❻.

When the program is run, if the rectangle of the ball overlaps the rectangle of the target, the message is written to the shell window. If you move the ball away from the target, the message stops being written out.

Dealing with Repeating Keys in Continuous Mode

The second way to handle keyboard interactions in pygame is to *poll* the keyboard. This involves asking pygame for a list representing which keys are currently down in every frame using the following call:

```
<aTuple> = pygame.key.get_pressed()
```

This call returns a tuple of 0s and 1s representing the state of each key: 0 if the key is up, 1 if the key is down. You can then use constants defined within pygame as an index into the returned tuple to see if a *particular* key is down. For example, the following lines can be used to determine the state of the A key:

```
keyPressedTuple = pygame.key.get_pressed()
# Now use a constant to get the appropriate element of the tuple
aIsDown = keyPressedTuple[pygame.K_a]
```

The full listing of constants representing all keys defined in pygame can be found at *https://www.pygame.org/docs/ref/key.html*.

The code in Listing 5-5 shows how we can use this technique to move an image continuously rather than once per key press. In this version, we move the keyboard handling from section #7 to section #8. The rest of the code is identical to the previous version in Listing 5-4.

File: PygameDemo3_MoveByKeyboard/PygameMoveByKeyboardContinuous.py

```
# pygame demo 3(b) - one image, continuous mode, move as long as a key is down

--- snip ---
    # 7 - Check for and handle events
    for event in pygame.event.get():
        # Clicked the close button? Quit pygame and end the program
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # 8 - Do any "per frame" actions
    # Check for user pressing keys
 ❶ keyPressedTuple = pygame.key.get_pressed()

    if keyPressedTuple[pygame.K_LEFT]:  # moving left
        ballX = ballX - N_PIXELS_TO_MOVE

    if keyPressedTuple[pygame.K_RIGHT]:  # moving right
        ballX = ballX + N_PIXELS_TO_MOVE

    if keyPressedTuple[pygame.K_UP]:  # moving up
        ballY = ballY - N_PIXELS_TO_MOVE

    if keyPressedTuple[pygame.K_DOWN]:  # moving down
        ballY = ballY + N_PIXELS_TO_MOVE

    # Check if the ball is colliding with the target
    ballRect = pygame.Rect(ballX, ballY,
                           BALL_WIDTH_HEIGHT, BALL_WIDTH_HEIGHT)
    if ballRect.colliderect(targetRect):
        print('Ball is touching the target')
--- snip ---
```

Listing 5-5: Handling keys being held down

The keyboard-handling code in Listing 5-5 does not rely on events, so we place the new code outside of the `for` loop that iterates through all events returned by pygame ❶.

Because we are doing this check in every frame, the movement of the ball will appear to be continuous as long as the user holds down a key. For example, if the user presses and holds the right arrow key, this code will add 3 to the value of the `ballX` coordinate in every frame, and the user will see the ball moving smoothly to the right. When they stop pressing the key, the movement stops.

The other change is that this approach allows you to check for multiple keys being down at the same time. For example, if the user presses and holds the left and down arrow keys, the ball will move diagonally down and to the left. You can check for as many keys being held down as you wish. However, the number of *simultaneous* key presses that can be detected is limited by the operating system, the keyboard hardware, and many other factors. The typical limit is around four keys, but your mileage may vary.

Creating a Location-Based Animation

Next, we'll build a location-based animation. This code will allow us to move an image diagonally and then have it appear to bounce off the edges of the window. This was a favorite technique of screensavers on old CRT-based monitors, to avoid burning in a static image.

We'll change the location of our image slightly in every frame. We'll also check if the result of that movement would place any part of the image outside one of the window boundaries and, if so, reverse the movement in that direction. For example, if the image was moving down and would cross the bottom of the window, we would reverse the direction and make the image start moving up.

We'll again use the same starting template. Listing 5-6 gives the full source code.

File: PygameDemo4_OneBallBounce/PygameOneBallBounceXY.py

```
# pygame demo 4(a) - one image, bounce around the window using (x, y) coords

# 1 - Import packages
import pygame
from pygame.locals import *
import sys
import random

# 2 - Define constants
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30
BALL_WIDTH_HEIGHT = 100
N_PIXELS_PER_FRAME = 3

# 3 - Initialize the world
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

# 4 - Load assets: image(s), sound(s), etc.
ballImage = pygame.image.load('images/ball.png')

# 5 - Initialize variables
```

```
MAX_WIDTH = WINDOW_WIDTH - BALL_WIDTH_HEIGHT
MAX_HEIGHT = WINDOW_HEIGHT - BALL_WIDTH_HEIGHT
❶ ballX = random.randrange(MAX_WIDTH)
ballY = random.randrange(MAX_HEIGHT)
xSpeed = N_PIXELS_PER_FRAME
ySpeed = N_PIXELS_PER_FRAME

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        # Clicked the close button? Quit pygame and end the program
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # 8 - Do any "per frame" actions
  ❷ if (ballX < 0) or (ballX >= MAX_WIDTH):
        xSpeed = -xSpeed  # reverse X direction

    if (ballY < 0) or (ballY >= MAX_HEIGHT):
        ySpeed = -ySpeed  # reverse Y direction

    # Update the ball's location, using the speed in two directions
  ❸ ballX = ballX + xSpeed
    ballY = ballY + ySpeed

    # 9 - Clear the window before drawing it again
    window.fill(BLACK)

    # 10 - Draw the window elements
    window.blit(ballImage, (ballX, ballY))

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)
```

Listing 5-6: A location-based animation, bouncing a ball around the window

We start by creating and initializing the two variables `xSpeed` and `ySpeed` ❶, which determine how far and in what direction the image should move in each frame. We initialize both variables to the number of pixels to move per frame (3), so the image will start by moving three pixels to the right (the positive x direction) and three pixels down (the positive y direction).

In the key part of the program, we handle the x- and y-coordinates separately ❷. First, we check to see if the x-coordinate of the ball is less than zero, meaning that part of the image is off the left edge, or past the `MAX_WIDTH` pixel and so effectively off the right edge. If either of these is the case, we reverse the sign of the speed in the x direction, meaning it will go in the opposite direction. For example, if the ball was moving to the right

and went off the right edge, we would change the value of `xSpeed` from `3` to `-3` to cause the ball to start moving to the left, and vice versa.

Then we do a similar check for the y-coordinate to make the ball bounce off the top or bottom edge, as needed.

Finally, we update the position of the ball by adding the `xSpeed` to the `ballX` coordinate and adding the `ySpeed` to the `ballY` coordinate ❸. This positions the ball at a new location on both axes.

At the bottom of the main loop, we draw the ball. Since we're updating the values of `ballX` and `ballY` in every frame, the ball appears to animate smoothly. Try it out. Whenever the ball reaches any edge, it seems to bounce off.

Using Pygame rects

Next I'll present a different way to achieve the same result. Rather than keeping track of the current x- and y-coordinates of the ball in separate variables, we'll use the `rect` of the ball, update the `rect` every frame, and check if performing the update would cause any part of the `rect` to move outside an edge of the window. This results in fewer variables, and because we'll start by making a call to get the `rect` of an image, it will work with images of any size.

When you create a `rect` object, in addition to remembering the `left`, `top`, `width`, and `height` as attributes of the rectangle, that object also calculates and maintains a number of other attributes for you. You can access any of these attributes directly by name using *dot syntax*, as shown in Table 5-1. (I'll provide more detail on this in Chapter 8.)

Table 5-1: Direct Access to Attributes of a `rect`

Attribute	Description
`<rect>.x`	The x-coordinate of the left edge of the `rect`
`<rect>.y`	The y-coordinate of the top edge of the `rect`
`<rect>.left`	The x-coordinate of the left edge of the `rect` (same as `<rect>.x`)
`<rect>.top`	The y-coordinate of the top edge of the `rect` (same as `<rect>.y`)
`<rect>.right`	The x-coordinate of the right edge of the `rect`
`<rect>.bottom`	The y-coordinate of the bottom edge of the `rect`
`<rect>.topleft`	A two-integer tuple: the coordinates of the upper-left corner of the `rect`
`<rect>.bottomleft`	A two-integer tuple: the coordinates of the lower-left corner of the `rect`
`<rect>.topright`	A two-integer tuple: the coordinates of the upper-right corner of the `rect`
`<rect>.bottomright`	A two-integer tuple: the coordinates of the lower-right corner of the `rect`

(continued)

Table 5-1: Direct Access to Attributes of a `rect` *(continued)*

Attribute	Description
`<rect>.midtop`	A two-integer tuple: the coordinates of the midpoint of the top edge of the rect
`<rect>.midleft`	A two-integer tuple: the coordinates of the midpoint of the left edge of the rect
`<rect>.midbottom`	A two-integer tuple: the coordinates of the midpoint of the bottom edge of the rect
`<rect>.midright`	A two-integer tuple: the coordinates of the midpoint of the right edge of the rect
`<rect>.center`	A two-integer tuple: the coordinates at the center of the rect
`<rect>.centerx`	The x-coordinate of the center of the width of the rect
`<rect>.centery`	The y-coordinate of the center of the height of the rect
`<rect>.size`	A two-integer tuple: the (width, height) of the rect
`<rect>.width`	The width of the rect
`<rect>.height`	The height of the rect
`<rect>.w`	The width of the rect (same as `<rect>.width`)
`<rect>.h`	The height of the rect (same as `<rect>.height`)

A pygame `rect` also can be thought of, and accessed as, a list of four elements. Specifically, you can use an index to get or set any individual part of a `rect`. For instance, using the `ballRect`, the individual elements can be accessed as:

- `ballRect[0]` is the x value (but you could also use `ballRect.left`)
- `ballRect[1]` is the y value (but you could also use `ballRect.top`)
- `ballRect[2]` is the width (but you could also use `ballRect.width`)
- `ballRect[3]` is the height (but you could also use `ballRect.height`)

Listing 5-7 is an alternative version of our bouncing ball program that maintains all the information about the ball in a rectangle object.

File: PygameDemo4_OneBallBounce/PygameOneBallBounceRects.py

```
# pygame demo 4(b) - one image, bounce around the window using rects

# 1 - Import packages
import pygame
from pygame.locals import *
import sys
import random

# 2 - Define constants
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
```

```
FRAMES_PER_SECOND = 30
N_PIXELS_PER_FRAME = 3

# 3 - Initialize the world
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

# 4 - Load assets: image(s), sound(s), etc.
ballImage = pygame.image.load('images/ball.png')

# 5 - Initialize variables
❶ ballRect = ballImage.get_rect()
MAX_WIDTH = WINDOW_WIDTH - ballRect.width
MAX_HEIGHT = WINDOW_HEIGHT - ballRect.height
ballRect.left = random.randrange(MAX_WIDTH)
ballRect.top = random.randrange(MAX_HEIGHT)
xSpeed = N_PIXELS_PER_FRAME
ySpeed = N_PIXELS_PER_FRAME

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        # Clicked the close button? Quit pygame and end the program
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # 8 - Do any "per frame" actions
  ❷ if (ballRect.left < 0) or (ballRect.right >= WINDOW_WIDTH):
        xSpeed = -xSpeed  # reverse X direction

    if (ballRect.top < 0) or (ballRect.bottom >= WINDOW_HEIGHT):
        ySpeed = -ySpeed  # reverse Y direction

    # Update the ball's rectangle using the speed in two directions
    ballRect.left = ballRect.left + xSpeed
    ballRect.top = ballRect.top + ySpeed

    # 9 - Clear the window before drawing it again
    window.fill(BLACK)

    # 10 - Draw the window elements
  ❸ window.blit(ballImage, ballRect)

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)
```

Listing 5-7: A location-based animation, bouncing a ball around the window, using `rects`

This approach of using a `rect` object is neither better nor worse than using separate variables. The resulting program works exactly the same as the original. The important lesson here is how you can use and manipulate attributes of a `rect` object.

After loading the image of the ball, we call the `get_rect()` method ❶ to get the bounding rectangle of the image. That call returns a `rect` object, which we store into a variable called `ballRect`. We use `ballRect.width` and `ballRect.height` to get direct access to the width and height of the ball image. (In the previous version, we used a constant of 100 for the width and the height.) Getting these values from the image that was loaded makes our code much more adaptable because it means we can use a graphic of any size.

The code also uses the attributes of the rectangle rather than using separate variables for checking if any part of the ball's rectangle goes over an edge. We can use `ballRect.left` and `ballRect.right` to see if the `ballRect` is off the left or right edges ❷. We do a similar test with `ballRect.top` and `ballRect.bottom`. Rather than updating individual x- and y-coordinate variables, we update the `left` and `top` of the `ballRect`.

The other subtle but important change is in the call to draw the ball ❸. The second argument in the call to `blit()` can be either an (x, y) tuple or a `rect`. The code inside `blit()` uses the left and top position in the `rect` as the x- and y-coordinates.

Playing Sounds

There are two types of sounds that you might want to play in your programs: short sound effects and background music.

Playing Sound Effects

All sound effects must live in external files and must be in either *.wav* or *.ogg* format. Playing a relatively short sound effect consists of two steps: load the sound from an external sound file once; then at the appropriate time(s) play your sound.

To load a sound effect into memory, you use a line like this:

```
<soundVariable> = pygame.mixer.Sound(<path to sound file>)
```

To play the sound effect, you only need to call its `play()` method:

```
<soundVariable>.play()
```

We'll modify Listing 5-7 to add a "boing" sound effect whenever the ball bounces off a side of the window. There is a *sounds* folder in the project folder at the same level as the main program. Right after loading the ball image, we load the sound file by adding this code:

```
# 4 - Load assets: image(s), sound(s), etc.
ballImage = pygame.image.load('images/ball.png')
bounceSound = pygame.mixer.Sound('sounds/boing.wav')
```

To play the "boing" sound effect whenever we change either the horizontal or vertical direction of the ball, we modify section #8 to look like this:

```
# 8 - Do any "per frame" actions
    if (ballRect.left < 0) or (ballRect.right >= WINDOW_WIDTH):
        xSpeed = -xSpeed  # reverse X direction
        bounceSound.play()

    if (ballRect.top < 0) or (ballRect.bottom >= WINDOW_HEIGHT):
        ySpeed = -ySpeed  # reverse Y direction
        bounceSound.play()
```

When you find a condition that should play a sound effect, you add a call to the `play()` method of the sound. There are many more options for controlling sound effects; you can find details in the official documentation at *https://www.pygame.org/docs/ref/mixer.html*.

Playing Background Music

Playing background music involves two lines of code using calls to the `pygame.mixer.music` module. First, you need this to load the sound file into memory:

```
pygame.mixer.music.load(<path to sound file>)
```

The *<path to sound file>* is a path string where the sound file can be found. You can use *.mp3* files, which seem to work best, as well as *.wav* or *.ogg* files. When you want to start the music playing, you need to make this call:

```
pygame.mixer.music.play(<number of loops>, <starting position>)
```

To play some background music repeatedly, you can pass in a `-1` for *<number of loops>* to run the music forever. The *<starting position>* is typically set to `0` to indicate that you want to play the sound from the beginning.

There is a downloadable, modified version of the bouncing ball program that properly loads the sound effect and background music files and starts the background sound playing. The only changes are in section #4, as shown here.

File: PygameDemo4_OneBallBounce/PyGameOneBallBounceWithSound.py

```
# 4 - Load assets: image(s), sound(s), etc.
ballImage = pygame.image.load('images/ball.png')
bounceSound = pygame.mixer.Sound('sounds/boing.wav')
pygame.mixer.music.load('sounds/background.mp3')
pygame.mixer.music.play(-1, 0.0)
```

Pygame allows for much more intricate handling of background sounds. You can find the full documentation at *https://www.pygame.org/docs/ref/music.html#module-pygame.mixer.music*.

NOTE *In order to make future examples more clearly focused on OOP, I'll leave out calls to play sound effects and background music. But adding sounds greatly enhances the user experience of a game, and I strongly encourage including them.*

Drawing Shapes

Pygame offers a number of built-in functions that allow you to draw certain shapes known as *primitives*, which include lines, circles, ellipses, arcs, polygons, and rectangles. Table 5-2 provides a list of these functions. Note that there are two calls that draw *anti-aliased* lines. These are lines that include blended colors at the edges to make the lines look smooth and less jagged. There are two key advantages to using these drawing functions: they execute extremely quickly, and they allow you to draw simple shapes without having to create or load images from external files.

Table 5-2: Functions for Drawing Shapes

Function	Description
`pygame.draw.aaline()`	Draws an anti-aliased line
`pygame.draw.aalines()`	Draws a series of anti-aliased lines
`pygame.draw.arc()`	Draws an arc
`pygame.draw.circle()`	Draws a circle
`pygame.draw.ellipse()`	Draws an ellipse
`pygame.draw.line()`	Draws a line
`pygame.draw.lines()`	Draws a series of lines
`pygame.draw.polygon()`	Draws a polygon
`pygame.draw.rect()`	Draws a rectangle

Figure 5-7 shows the output of a sample program that demonstrates calls to these primitive drawing functions.

Listing 5-8 is the code of the sample program, using the same 12-step template that produced the output in Figure 5-7.

File: PygameDemo5_DrawingShapes.py

```
# pygame demo 5 - drawing

--- snip ---
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        # Clicked the close button? Quit pygame and end the program
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

```
    # 8 - Do any "per frame" actions

    # 9 - Clear the window
    window.fill(GRAY)

❶   # 10 - Draw all window elements
    # Draw a box
    pygame.draw.line(window, BLUE, (20, 20), (60, 20), 4)  # top
    pygame.draw.line(window, BLUE, (20, 20), (20, 60), 4)  # left
    pygame.draw.line(window, BLUE, (20, 60), (60, 60), 4)  # right
    pygame.draw.line(window, BLUE, (60, 20), (60, 60), 4)  # bottom
    # Draw an X in the box
    pygame.draw.line(window, BLUE, (20, 20), (60, 60), 1)
    pygame.draw.line(window, BLUE, (20, 60), (60, 20), 1)

    # Draw a filled circle and an empty circle
    pygame.draw.circle(window, GREEN, (250, 50), 30, 0) # filled
    pygame.draw.circle(window, GREEN, (400, 50), 30, 2) # 2 pixel edge

    # Draw a filled rectangle and an empty rectangle
    pygame.draw.rect(window, RED, (250, 150, 100, 50), 0) # filled
    pygame.draw.rect(window, RED, (400, 150, 100, 50), 1) # 1 pixel edge

    # Draw a filled ellipse and an empty ellipse
    pygame.draw.ellipse(window, YELLOW, (250, 250, 80, 40), 0) # filled
    pygame.draw.ellipse(window, YELLOW, (400, 250, 80, 40), 2) # 2 pixel edge

    # Draw a six-sided polygon
    pygame.draw.polygon(window, TEAL, ((240, 350), (350, 350),
                                       (410, 410), (350, 470),
                                       (240, 470), (170, 410)))

    # Draw an arc
    pygame.draw.arc(window, BLUE, (20, 400, 100, 100), 0, 2, 5)

    # Draw anti-aliased lines: a single line, then a list of points
    pygame.draw.aaline(window, RED, (500, 400),  (540, 470), 1)
    pygame.draw.aalines(window, BLUE, True,
                        ((580, 400), (587, 450),
                         (595, 460), (600, 444)), 1)

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)  # make pygame wait
```

Listing 5-8: A program to demonstrate calls to primitive drawing functions in pygame

The drawing of all the primitives occurs in section #10 ❶. We make calls to pygame's drawing functions to draw a box with two diagonals, filled and empty circles, filled and empty rectangles, filled and empty ovals, a six-sided polygon, an arc, and two anti-aliased lines.

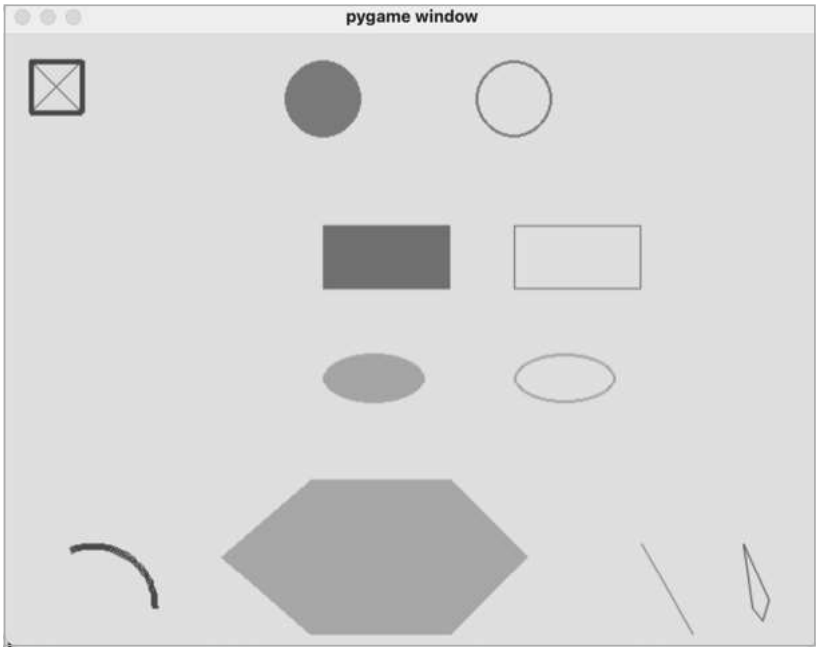

Figure 5-7: A sample program that demonstrates using calls to draw primitive shapes

Reference for Primitive Shapes

For your reference, here is the documentation for the pygame methods to draw these primitives. In all of the following, the `color` argument expects you to pass in a tuple of RGB values:

Anti-aliased line

```
pygame.draw.aaline(window, color, startpos, endpos, blend=True)
```

Draws an anti-aliased line in the window. If `blend` is `True`, the shades will be blended with existing pixel shades instead of overwriting pixels.

Anti-aliased lines

```
pygame.draw.aalines(window, color, closed, points, blend=True)
```

Draws a sequence of anti-aliased lines in the window. The `closed` argument is a simple Boolean; if it's `True`, a line will be drawn between the first and last points to complete the shape. The `points` argument is a list or tuple of (x, y) coordinates to be connected by line segments (there must be at least two). The Boolean `blend` argument, if set to `True`, will blend the shades with existing pixel shades instead of overwriting them.

Arc

```
pygame.draw.arc(window, color, rect, angle_start, angle_stop, width=0)
```

Draws an arc in the window. The arc will fit inside the given `rect`. The two `angle` arguments are the initial and final angles (in radians, with zero on the right). The `width` argument is the thickness to draw the outer edge.

Circle

```
pygame.draw.circle(window, color, pos, radius, width=0)
```

Draws a circle in the window. The `pos` is the center of the circle, and `radius` is the radius. The `width` argument is the thickness to draw the outer edge. If `width` is `0`, then the circle will be filled.

Ellipse

```
pygame.draw.ellipse(window, color, rect, width=0)
```

Draws an ellipse in the window. The given `rect` is the area that the ellipse will fill. The `width` argument is the thickness to draw the outer edge. If `width` is `0`, then the ellipse will be filled.

Line

```
pygame.draw.line(window, color, startpos, endpos, width=1)
```

Draws a line in a window. The `width` argument is the thickness of the line.

Lines

```
pygame.draw.lines(window, color, closed, points, width=1)
```

Draws a sequence of lines in the window. The `closed` argument is a simple Boolean; if it's `True`, a line will be drawn between the first and last points to complete the shape. The `points` argument is a list or tuple of (x, y) coordinates to be connected by line segments (there must be at least two). The `width` argument is the thickness of the line. Note that specifying a line width wider than 1 does not fill in the gaps between the lines. Therefore, wide lines and sharp corners won't be joined seamlessly.

Polygon

```
pygame.draw.polygon(window, color, pointslist, width=0)
```

Draws a polygon in the window. The `pointslist` specifies the vertices of the polygon. The `width` argument is the thickness to draw the outer edge. If `width` is `0`, then the polygon will be filled.

Rectangle

```
pygame.draw.rect(window, color, rect, width=0)
```

Draws a rectangle in the window. The `rect` is the area of the rectangle. The `width` argument is the thickness to draw the outer edge. If `width` is `0`, then the rectangle will be filled.

NOTE *For additional information, see* http://www.pygame.org/docs/ref/draw.html.

The set of primitive calls allows you the flexibility to draw any shapes you wish. Again, the order in which you make calls is important. Think of the order of your calls as layers; elements that are drawn early can be overlaid by later calls to any other drawing primitive function.

Summary

In this chapter I introduced the basics of pygame. You installed pygame on your computer, then learned about the model of event-driven programming and the use of events, which is very different from coding text-based programs. I explained the coordinate system of pixels in a window and the way that colors are represented in code.

To start right at the beginning with pygame, I introduced a 12-section template that does nothing but bring up a window and can be used to build any pygame-based program. Using that framework, we then built sample programs that showed how to draw an image in the window (using `blit()`), how to detect mouse events, and how to handle keyboard input. The next demonstration explained how to build a location-based animation.

Rectangles are highly important in pygame, so I covered how the attributes of a `rect` object can be used. I also provided some example code to show how to play sound effects and background music to enhance the user's enjoyment of your programs. Finally, I introduced how to use pygame methods to draw primitive shapes in a window.

While I have introduced many concepts within pygame, almost everything I showed in this chapter has essentially been procedural. The `rect` object is an example of object-oriented code built directly into pygame. In the next chapter, I'll show how to use OOP in code to use pygame more effectively.

6

OBJECT-ORIENTED PYGAME

In this chapter I'll demonstrate how you can use OOP techniques effectively within the pygame framework. We'll start off with an example of procedural code, then split that code into a single class and some main code that calls the methods of that class. After that, we'll build two classes, `SimpleButton` and `SimpleText`, that implement basic user interface widgets: a button and a field for displaying text. I'll also introduce the concept of a callback.

Building the Screensaver Ball with OOP Pygame

In Chapter 5, we created an old-school screensaver where a ball bounced around inside a window (Listing 5-6, if you need to refresh your memory).

That code works, but the data for the ball and the code to manipulate the ball are intertwined, meaning there's a lot of initialization code, and the code to update and draw the ball are embedded in the 12-step framework.

A more modular approach is to split the code into a `Ball` class and a main program that instantiates a `Ball` object and makes calls to its methods. In this section we'll make this split, and I'll show you how to create multiple balls from the `Ball` class.

Creating a Ball Class

We'll start by extracting all code relating to the ball from the main program and moving it into a separate `Ball` class. Looking at the original code, we can see that the sections that deal with the ball are:

- Section #4, which loads the image of the ball
- Section #5, which creates and initializes all the variables that have something to do with the ball
- Section #8, which includes code for moving the ball, detecting an edge bounce, and changing speed and direction
- Section #10, which draws the ball

From this we can conclude that our `Ball` class will require the following methods:

`create()` Loads an image, sets a location, and initializes all instance variables

`update()` Changes the location of the ball in every frame, based on the x speed and y speed of the ball

`draw()` Draws the ball in the window

The first step is to create a project folder, in which you need a *Ball.py* for the new `Ball` class, the main code file *Main_BallBounce.py*, and an *images* folder containing the *ball.png* image file.

Listing 6-1 shows the code of the new `Ball` class.

File: PygameDemo6_BallBounceObjectOriented/Ball.py

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

# Ball class
class Ball():

  ❶ def __init__(self, window, windowWidth, windowHeight):
        self.window = window  # remember the window, so we can draw later
        self.windowWidth = windowWidth
        self.windowHeight = windowHeight
```

```
    ❷ self.image = pygame.image.load('images/ball.png')
      # A rect is made up of [x, y, width, height]
      ballRect = self.image.get_rect()
      self.width = ballRect.width
      self.height = ballRect.height
      self.maxWidth = windowWidth - self.width
      self.maxHeight = windowHeight - self.height

      # Pick a random starting position
    ❸ self.x = random.randrange(0, self.maxWidth)
      self.y = random.randrange(0, self.maxHeight)

      # Choose a random speed between -4 and 4, but not zero,
      # in both the x and y directions
    ❹ speedsList = [-4, -3, -2, -1, 1, 2, 3, 4]
      self.xSpeed = random.choice(speedsList)
      self.ySpeed = random.choice(speedsList)

    ❺ def update(self):
      # Check for hitting a wall. If so, change that direction.
      if (self.x < 0) or (self.x >= self.maxWidth):
          self.xSpeed = -self.xSpeed

      if (self.y < 0) or (self.y >= self.maxHeight):
          self.ySpeed = -self.ySpeed

      # Update the Ball's x and y, using the speed in two directions
      self.x = self.x + self.xSpeed
      self.y = self.y + self.ySpeed

  ❻ def draw(self):
      self.window.blit(self.image, (self.x, self.y))
```

Listing 6-1: The new `Ball` class

When we instantiate a `Ball` object, the `__init__()` method receives three pieces of data: the window to draw into, the width of the window, and the height of the window ❶. We save the `window` variable into the instance variable `self.window` so that we can use it later in the `draw()` method, and we do the same with the `self.windowHeight` and `self.windowWidth` instance variables. We then load the image of the ball using the path to the file and get the `rect` of that ball image ❷. We need the `rect` to calculate the maximum values for `x` and `y` so that the ball will always fully appear in the window. Next, we pick a randomized starting location for the ball ❸. Finally, we set the speed in the x and y directions to a random value between –4 and 4 (but not 0), representing the number of pixels to move per frame ❹. Because of these numbers, the ball may move differently each time we run the program. All these values are saved in instance variables to be used by other methods.

In the main program, we'll call the `update()` method in each frame of the main loop, so this is where we place the code that checks for the ball

hitting any border of the window ❺. If it does hit an edge, we reverse the speed in that direction and modify the x- and y-coordinates (`self.x` and `self.y`) by the current speed in the x and y directions.

We'll also call the `draw()` method, which simply calls `blit()` to draw the ball at its current x- and y-coordinates ❻, in every frame of the main loop.

Using the Ball Class

Now all functionality associated with a ball has been placed in the `Ball` class code. All the main program needs to do is create the ball, then call its `update()` and `draw()` methods in every frame. Listing 6-2 shows the greatly simplified code of the main program.

File: PygameDemo6_BallBounceObjectOriented/Main_BallBounce.py

```
# pygame demo 6(a) - using the Ball class, bounce one ball

# 1 - Import packages
import pygame
from pygame.locals import *
import sys
import random
❶ from Ball import *  # bring in the Ball class code

# 2 - Define constants
BLACK = (0, 0, 0)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30

# 3 - Initialize the world
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

# 4 - Load assets: image(s), sound(s), etc.

# 5 - Initialize variables
❷ oBall = Ball(window, WINDOW_WIDTH, WINDOW_HEIGHT)

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # 8 - Do any "per frame" actions
  ❸ oBall.update()  # tell the Ball to update itself
```

```
    # 9 - Clear the window before drawing it again
    window.fill(BLACK)

    # 10 - Draw the window elements
❹ oBall.draw()  # tell the Ball to draw itself

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)
```

Listing 6-2: The new main program that instantiates a `Ball` and makes calls to its methods

If you compare this new main program with the original code in Listing 5-6, you'll see that it's much simpler and clearer. We use an `import` statement to bring in the `Ball` class code ❶. We create a `Ball` object, passing in the window that we created and the width and height of that window ❷, and we save the resulting `Ball` object in a variable named `oBall`.

The responsibility of moving the ball is now in the `Ball` class code, so here we only need to call the `update()` method of the `oBall` object ❸. Since the `Ball` object knows how big the window is, how big the image of the ball is, and the location and the speed of the ball, it can do all the calculations it needs to do to move the ball and bounce it off the walls.

The main code calls the `draw()` method of the `oBall` object ❹, but the actual drawing is done in the `oBall` object.

Creating Many Ball Objects

Now let's make a slight but important modification to the main program to create multiple `Ball` objects. This is one of the real powers of object orientation: to create three balls, we only have to instantiate three `Ball` objects from the `Ball` class. Here we'll use a basic approach and build a list of `Ball` objects. In each frame, we'll iterate through the list of `Ball` objects, tell each one to update its location, then iterate again to tell each one to draw itself. Listing 6-3 shows a modified main program that creates and updates three `Ball` objects.

File: PygameDemo6_BallBounceObjectOriented/Main_BallBounceManyBalls.py

```
# pygame demo 6(b) - using the Ball class, bounce many balls

--- snip ---
N_BALLS = 3
--- snip ---

# 5 - Initialize variables
❶ ballList = []
for oBall in range(0, N_BALLS):
    # Each time through the loop, create a Ball object
    oBall = Ball(window, WINDOW_WIDTH, WINDOW_HEIGHT)
```

```
    ballList.append(oBall)  # append the new Ball to the list of Balls

# 6 - Loop forever
while True:

    --- snip ---

    # 8 - Do any "per frame" actions
❷   for oBall in ballList:
        oBall.update()  # tell each Ball to update itself

    # 9 - Clear the window before drawing it again
    window.fill(BLACK)

    # 10 - Draw the window elements
❸   for oBall in ballList:
        oBall.draw()   # tell each Ball to draw itself

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)
```

Listing 6-3: Creating, moving, and displaying three balls

We start with an empty list of `Ball` objects ❶. Then we have a loop that creates three `Ball` objects, each of which we append to our list of `Ball` objects, `ballList`. Each `Ball` object chooses and remembers a randomized starting location and a randomized speed in both the x and y directions.

Inside the main loop, we iterate through all the `Ball` objects and tell each one to update itself ❷, changing the x- and y-coordinates of each `Ball` object to a new location. We then iterate through the list again, calling the `draw()` method of each `Ball` object ❸.

When we run the program, we see three balls, each starting at a randomized location and each moving with a randomized x and y speed. Each ball bounces correctly off the boundaries of the window.

Using this object-oriented approach, we made no changes to the `Ball` class, but just changed our main program to now manage a list of `Ball` objects instead of a single `Ball` object. This is a common, and very positive, side effect of OOP code: well-written classes can often be reused without change.

Creating Many, Many Ball Objects

We can change the value of the constant `N_BALLS` from `3` to some much larger value, like `300`, to quickly create that many balls (Figure 6-1). By changing just a single constant, we make a major change to the behavior of the program. Each ball maintains its own speed and location and draws itself.

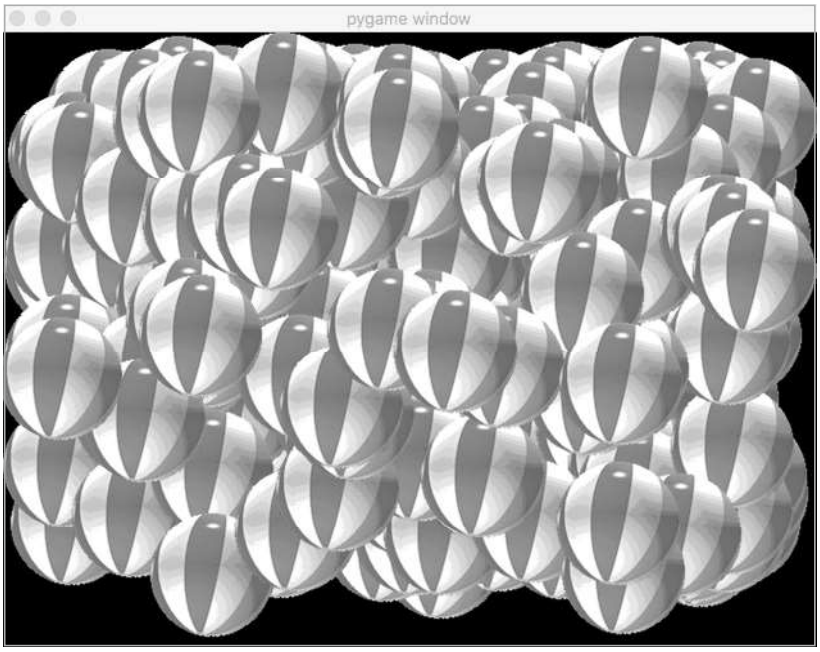

Figure 6-1: Creating, updating, and drawing 300 `Ball` objects

The fact that we can instantiate any number of objects from a single script will be vital not only in defining game objects like spaceships, zombies, bullets, treasures, and so on, but also in building GUI controls such as buttons, checkboxes, text input fields, and text outputs.

Building a Reusable Object-Oriented Button

The simple button is one of the most recognizable elements of a graphical user interface. The standard behavior of a button consists of the user using their mouse to click down on the button image and then releasing it.

Buttons typically consist of at least two images: one to represent the *up* or normal state of the button and another to represent the *down* or pressed state of the button. The sequence of a click can be broken down into the following steps:

1. User moves the mouse pointer over the button
2. User presses the mouse button down
3. Program reacts by changing the image to the down state
4. User releases the mouse button
5. Program reacts by showing the up image of the button
6. Program performs some action based on the button click

Good GUIs also allow the user to click down on a button, temporarily roll off the button, changing the button to the up state, and then, with the

mouse button still down, roll back over the image so the button changes back to the down image. If the user clicks down on a button but then rolls the mouse off and lifts up on the mouse button, that is not considered a click. This means the program takes action only when the user presses down and releases while the mouse is positioned over the image of a button.

Building a Button Class

The button behavior should be common and consistent for all buttons used in a GUI, so we'll build a class that takes care of the behavior details. Once we've built a simple button class, we can instantiate any number of buttons and they'll all work exactly the same way.

Let's consider what behaviors our button class must support. We'll need methods to:

- Load the images of the up and down states, then initialize any instance variables needed to track the button's state.
- Tell the button about all events that the main program has detected and check whether there are any that the button needs to react to.
- Draw the current image representing the button.

Listing 6-4 presents the code of a `SimpleButton` class. (We'll build a more complicated button class in Chapter 7.) This class has three methods, `__init__()`, `handleEvent()`, and `draw()`, that implement the behaviors mentioned. The code of the `handleEvent()` method does get a little tricky, but once you have it working, it's incredibly easy to use. Feel free to work your way through it, but know that the implementation of the code is not that relevant. The important thing here is to understand the purpose and usage of the different methods.

File: PygameDemo7_SimpleButton/SimpleButton.py

```
# SimpleButton class
#
# Uses a "state machine" approach
#

import pygame
from pygame.locals import *

class SimpleButton():
    # Used to track the state of the button
    STATE_IDLE = 'idle' # button is up, mouse not over button
    STATE_ARMED = 'armed' # button is down, mouse over button
    STATE_DISARMED = 'disarmed' # clicked down on button, rolled off

    def __init__(self, window, loc, up, down): ❶
        self.window = window
        self.loc = loc
        self.surfaceUp = pygame.image.load(up)
```

```
        self.surfaceDown = pygame.image.load(down)

        # Get the rect of the button (used to see if the mouse is over the button)
        self.rect = self.surfaceUp.get_rect()
        self.rect[0] = loc[0]
        self.rect[1] = loc[1]

        self.state = SimpleButton.STATE_IDLE

    def handleEvent(self, eventObj): ❷
        # This method will return True if user clicks the button.
        # Normally returns False.

        if eventObj.type not in (MOUSEMOTION, MOUSEBUTTONUP, MOUSEBUTTONDOWN): ❸
            # The button only cares about mouse-related events
            return False

        eventPointInButtonRect = self.rect.collidepoint(eventObj.pos)

        if self.state == SimpleButton.STATE_IDLE:
            if (eventObj.type == MOUSEBUTTONDOWN) and eventPointInButtonRect:
                self.state = SimpleButton.STATE_ARMED

        elif self.state == SimpleButton.STATE_ARMED:
            if (eventObj.type == MOUSEBUTTONUP) and eventPointInButtonRect:
                self.state = SimpleButton.STATE_IDLE
                return True  # clicked!

            if (eventObj.type == MOUSEMOTION) and (not eventPointInButtonRect):
                self.state = SimpleButton.STATE_DISARMED

        elif self.state == SimpleButton.STATE_DISARMED:
            if eventPointInButtonRect:
                self.state = SimpleButton.STATE_ARMED
            elif eventObj.type == MOUSEBUTTONUP:
                self.state = SimpleButton.STATE_IDLE

        return False

    def draw(self): ❹
        # Draw the button's current appearance to the window
        if self.state == SimpleButton.STATE_ARMED:
            self.window.blit(self.surfaceDown, self.loc)

        else:  # IDLE or DISARMED
            self.window.blit(self.surfaceUp, self.loc)
```

Listing 6-4: The `SimpleButton` class

The `__init__()` method begins by saving all values passed in into instance variables ❶ to use in other methods. It then initializes a few more instance variables.

Whenever the main program detects any event, it calls the `handleEvent()` method ❷. This method first checks that the event is one of `MOUSEMOTION`,

MOUSEBUTTONUP, or MOUSEBUTTONDOWN ❸. The rest of the method is implemented as a *state machine*, a technique that I will go into more detail about in Chapter 15. The code is a little complicated, and you should feel free to study how it works, but for now note that it uses the instance variable self.state (over the course of multiple calls) to detect if the user has clicked on the button. The handleEvent() method returns True when the user completes a mouse click by pressing down on the button, then later releasing on the same button. In all other cases, handleEvent() returns False.

Finally, the draw() method uses the state of the object's instance variable self.state to decide which image (up or down) to draw ❹.

Main Code Using a SimpleButton

To use a SimpleButton in the main code, we first instantiate one from the SimpleButton class before the main loop starts with a line like this:

```
oButton = SimpleButton(window, (150, 30),
                       'images/buttonUp.png',
                       'images/buttonDown.png')
```

This line creates a SimpleButton object, specifying a location to draw it (as usual, the coordinates are for the top-left corner of the bounding rectangle) and providing the paths to both the up and down images of the button. In the main loop, any time any event happens we need to call the handleEvent() method to see if the user has clicked the button. If the user clicks the button, the program should perform some action. Also in the main loop, we need to call the draw() method to make the button show in the window.

We'll build a small test program, which will generate a user interface like Figure 6-2, to incorporate one instance of a SimpleButton.

Figure 6-2: The user interface of a program with a single instance of a SimpleButton

Whenever the user completes a click on the button, the program outputs a line of text in the shell saying that the button has been clicked. Listing 6-5 contains the main program code.

File: PygameDemo7_SimpleButton/Main_SimpleButton.py

```
# Pygame demo 7 - SimpleButton test

--- snip ---
# 5 - Initialize variables
# Create an instance of a SimpleButton
```

```
❶ oButton = SimpleButton(window, (150, 30),
                         'images/buttonUp.png',
                         'images/buttonDown.png')

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        # Pass the event to the button, see if it has been clicked on
      ❷ if oButton.handleEvent(event):
          ❸ print('User has clicked the button')

    # 8 - Do any "per frame" actions

    # 9 - Clear the window
    window.fill(GRAY)

    # 10 - Draw all window elements
  ❹ oButton.draw() # draw the button

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)
```

Listing 6-5: The main program that creates and reacts to a `SimpleButton`

Again, we start with the standard pygame template from Chapter 5. Before the main loop, we create an instance of our `SimpleButton` ❶, specifying a window to draw into, a location, a path to the up image, and a path to the down image.

Every time through the main loop, we need to react to events detected in the main program. To implement this, we call the `SimpleButton` class's `handleEvent()` method ❷ and pass in the `event` from the main program.

The `handleEvent()` method tracks all of the user's actions on the button (pressing down, releasing, rolling off, rolling back on). When `handleEvent()` returns `True`, indicating that a click has occurred, we perform the action associated with clicking that button. Here, we just print a message ❸.

Finally we call the button's `draw()` method ❹ to draw an image to represent the appropriate state of the button (up or down).

Creating a Program with Multiple Buttons

With our `SimpleButton` class, we can instantiate as many buttons as we wish. For example, we can modify our main program to incorporate three `SimpleButton` instances, as shown in Figure 6-3.

Figure 6-3: The main program with three `SimpleButton` objects

We don't need to make any changes to the `SimpleButton` class file to do this. We simply modify our main code to instantiate three `SimpleButton` objects instead of one.

File: PygameDemo7_SimpleButton/Main_SimpleButton3Buttons.py

```
oButtonA = SimpleButton(window, (25, 30),
                        'images/buttonAUp.png',
                        'images/buttonADown.png')
oButtonB = SimpleButton(window, (150, 30),
                        'images/buttonBUp.png',
                        'images/buttonBDown.png')
oButtonC = SimpleButton(window, (275, 30),
                        'images/buttonCUp.png',
                        'images/buttonCDown.png')
```

We now need to call the `handleEvent()` method of all three buttons:

```
        # Pass the event to each button, see if one has been clicked
        if oButtonA.handleEvent(event):
            print('User clicked button A.')
        elif oButtonB.handleEvent(event):
            print('User clicked button B.')
        elif oButtonC.handleEvent(event):
            print('User clicked button C.')
```

Finally, we tell each button to draw itself:

```
    oButtonA.draw()
    oButtonB.draw()
    oButtonC.draw()
```

When you run the program, you'll see a window with three buttons. Clicking any of the buttons prints a message showing the name of the button that was clicked.

The key idea here is that since we are using three instances of the same `SimpleButton` class, the behavior of each button will be identical. An important benefit of this approach is that any change to the code in the `SimpleButton` class will affect all buttons instantiated from the class. The main program does not need to worry about any details of the inner workings of the button code, needing only to call the `handleEvent()` method of each button in the main loop. Each button will return `True` or `False` to say that it has or has not been clicked.

Building a Reusable Object-Oriented Text Display

There are two different types of text in a pygame program: display text and input text. Display text is output from your program, equivalent to a call to the `print()` function, except it's displayed in a pygame window. Input text is string input from the user, equivalent to a call to `input()`. In this section, I'll discuss display text. We'll look at how to deal with input text in the next chapter.

Steps to Display Text

Displaying text in a window is a fairly complicated process in pygame because it's not simply displayed as a string in the shell, but requires you to choose a location, fonts and sizes, and other attributes. For example, you might use code like the following:

```
pygame.font.init()

myFont = pygame.font.SysFont('Comic Sans MS', 30)
textSurface = myfont.render('Some text', True, (0, 0, 0))
window.blit(textSurface, (10, 10))
```

We start by initializing the font system within pygame; we do this before the main loop starts. Then we tell pygame to load a particular font from the system by name. Here, we request Comic Sans with a font size of 30.

The next step is the key one: we use that font to *render* our text, which creates a graphical image of the text, called a *surface* in pygame. We supply the text we want to output, a Boolean that says whether we want our text to be anti-aliased, and a color in RGB format. Here, `(0, 0, 0)` indicates that we want our text to be black. Finally, using `blit()`, we draw the image of the text into the window at some (x, y) location.

This code works well to show the provided text in the window at the given location. However, if the text doesn't change, there will be a lot of wasted work done re-creating the `textSurface` on each iteration through the main loop. There are also a lot of details to remember, and you must get them all correct to draw the text properly. We can hide most of this complexity by building a class.

Creating a SimpleText Class

The idea is to build a set of methods that take care of font loading and text rendering in pygame, meaning we no longer have to remember the details of the implementation. Listing 6-6 contains a new class called `SimpleText` that does this work.

File: PygameDemo8_SimpleTextDisplay/SimpleText.py

```
# SimpleText class

import pygame
from pygame.locals import *
```

```
class SimpleText():

  ❶ def __init__(self, window, loc, value, textColor):
      ❷ pygame.font.init()
        self.window = window
        self.loc = loc
      ❸ self.font = pygame.font.SysFont(None, 30)
        self.textColor = textColor
        self.text = None # so that the call to setText below will
                         # force the creation of the text image
        self.setValue(value) # set the initial text for drawing

  ❹ def setValue(self, newText):
        if self.text == newText:
            return  # nothing to change

        self.text = newText  # save the new text
        self.textSurface = self.font.render(self.text, True, self.textColor)

  ❺ def draw(self):
        self.window.blit(self.textSurface, self.loc)
```

Listing 6-6: The `SimpleText` class for displaying text

You can think of a `SimpleText` object as a field in the window where you want text to be displayed. You can use one to display unchanging label text or to display text that changes throughout a program.

The `SimpleText` class has only three methods. The `__init__()` method ❶ expects the window to draw into, the location at which to draw the text in the window, any initial text you want to see displayed in the field, and a text color. Calling `pygame.font.init()` ❷ starts up pygame's font system. The call in the first instantiated `SimpleText` object actually does the initialization; any additional `SimpleText` objects will also make this call, but since fonts have already been initialized, the call returns immediately. We create a new `Font` object with `pygame.font.SysFont()` ❸. Rather than providing a specific font name, `None` indicates that we will use whatever the standard system font is.

The `setValue()` method renders an image of the text to display and saves that image in the `self.textSurface` instance variable ❹. As the program runs, any time you want to change the text that's displayed, you call the `setValue()` method, passing in the new text to display. The `setValue()` method has an optimization, too: it remembers the last text that it rendered, and before doing anything else, it checks if the new text is the same as the previous text. If the text has not changed, there is nothing to do and the method just returns. If there is new text, it renders the new text into a surface to be drawn.

The `draw()` method ❺ draws the image contained in the `self.textSurface` instance variable into the window at the given location. This method should be called in every frame.

There are multiple advantages to this approach:

- The class hides all the details of pygame's rendering of text, so the user of this class never needs to know what pygame-specific calls are needed to show text.

- Each `SimpleText` object remembers the window that it draws into, the location where the text should be placed, and the text color. Therefore, you only need to specify these values once, when you instantiate a `SimpleText` object, typically before the main loop starts.
- Each `SimpleText` object is also optimized to remember both the text that it was last told to draw and the image (`self.textSurface`) that it made from the current text. It only needs to render a new surface when the text changes.
- To show multiple pieces of text in a window, you only need to instantiate multiple `SimpleText` objects. This is a key concept of object-oriented programming.

Demo Ball with SimpleText and SimpleButton

To cap this off, we'll modify Listing 6-2 to use the `SimpleText` and `SimpleButton` classes. The updated program in Listing 6-7 keeps track of the number of times it goes through the main loop and reports that information at the top of the window. Clicking the Restart button resets the counter.

File: PygameDemo8_SimpleTextDisplay/Main_BallTextAndButton.py

```
# pygame demo 8 - SimpleText, SimpleButton, and Ball

# 1 - Import packages
import pygame
from pygame.locals import *
import sys
import random
❶ from Ball import *  # bring in the Ball class code
from SimpleText import *
from SimpleButton import *

# 2 - Define constants
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480
FRAMES_PER_SECOND = 30

# 3 - Initialize the world
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

# 4 - Load assets: image(s), sound(s), etc.

# 5 - Initialize variables
❷ oBall = Ball(window, WINDOW_WIDTH, WINDOW_HEIGHT)
oFrameCountLabel = SimpleText(window, (60, 20),
                          'Program has run through this many loops: ', WHITE)
```

```
oFrameCountDisplay = SimpleText(window, (500, 20), '', WHITE)
oRestartButton = SimpleButton(window, (280, 60),
                        'images/restartUp.png', 'images/restartDown.png')
frameCounter = 0

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

      ❸ if oRestartButton.handleEvent(event):
            frameCounter = 0  # clicked button, reset counter

    # 8 - Do any "per frame" actions
  ❹ oBall.update()  # tell the ball to update itself
    frameCounter = frameCounter + 1  # increment each frame
  ❺ oFrameCountDisplay.setValue(str(frameCounter))

   # 9 - Clear the window before drawing it again
    window.fill(BLACK)

    # 10 - Draw the window elements
  ❻ oBall.draw()  # tell the ball to draw itself
    oFrameCountLabel.draw()
    oFrameCountDisplay.draw()
    oRestartButton.draw()

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)
```

Listing 6-7: An example main program to show `Ball`, `SimpleText`, and `SimpleButton`

At the top of the program, we import the code of the `Ball`, `SimpleText`, and `SimpleButton` classes ❶. Before our main loop starts, we create an instance of the `Ball` ❷, two instances of the `SimpleText` class (`oFrameCountLabel` for the unchanging message label and `oFrameCountDisplay` for the changing display of frames), and an instance of the `SimpleButton` class that we store in `oRestartButton`. We also initialize a variable `frameCounter` to zero, which we will increment every time through the main loop.

In the main loop, we check if the user pressed the Restart button ❸. If `True`, we reset the frame counter.

We tell the ball to update its position ❹. We increment the frame counter, then call the `setValue()` method of the text field to show the new count of frames ❺. Finally, we tell the ball to draw itself tell the text fields to draw themselves, and tell the Restart button to draw itself, by calling the `draw()` method of each object ❻.

In the instantiation of the `SimpleText` objects, the last argument is a text color, and we specified that the objects should be rendered in `WHITE` so they can be seen against a `BLACK` background. In the next chapter, I'll show how to expand the `SimpleText` class to incorporate more attributes, without complicating the interface of the class. We'll build a more full-featured text object that has reasonable default values for each of these attributes, but allows you to override those defaults.

Interface vs. Implementation

The `SimpleButton` and `SimpleText` examples bring up the important topic of interface versus implementation. As mentioned in Chapter 4, the interface refers to how something is used, while the implementation refers to how something works (internally).

In an OOP environment, the interface is the set of methods in a class and their related parameters—also known as the *application programming interface (API)*. The implementation is the actual code of all the methods in the class.

An external package such as pygame will most likely come with documentation of the API that explains the calls that are available and the arguments you are expected to pass with each call. The full pygame API documentation is available at *https://www.pygame.org/docs/*.

When you write code that makes calls to pygame, you don't need to worry about the implementation of the methods you are using. For example, when you make a call to `blit()` to draw image, you really don't care *how* `blit()` does what it does; you just need to know *what* the call does and what arguments need to be passed in. On the other side, you can trust that the implementer(s) who wrote the `blit()` method have thought extensively about how to make `blit()` work most efficiently.

In the programming world, we often wear two hats as both the implementer and the application developer, so we need to make an effort to design APIs that not only make sense in the current situation, but also are general enough to be used by future programs of our own and by programs written by other people. Our `SimpleButton` and `SimpleText` classes are good examples, as they are written in a general way so that they can be reused easily. I'll talk more about interface versus implementation in Chapter 8, when we look at encapsulation.

Callbacks

When using a `SimpleButton` object, we handle checking for and reacting to a button click like this:

```
if oButton.handleEvent(event):
    print('The button was clicked')
```

This approach to handling events works well with the `SimpleButton` class. However, some other Python packages and many other programming languages handle events in a different way: with a *callback*.

callback A function or method of an object that is called when a particular action, event, or condition happens.

An easy way to understand this is to think about the 1984 hit movie *Ghostbusters*. The tagline for the movie is "Who you gonna call?" In the movie, the Ghostbusters ran an ad on TV that told people that if they saw a ghost (that's the event to look for), they should call the Ghostbusters (the callback) to get rid of it. Upon receiving the call, the Ghostbusters take the appropriate actions to eliminate the ghost.

As an example, consider a button object that is initialized to have a callback. When the user clicks the button, the button will call the callback function or method. That function or method executes whatever code is needed to react to the button click.

Creating a Callback

To set up a callback, when you create an object or call one of an object's methods, you pass the name of a function or a method of an object to be called. As an example, there is a standard GUI package for Python called `tkinter`. The code needed to create a button with this package is very different from what I have shown—here's an example:

```
import tkinter

def myFunction():
    print('myCallBackFunction was called')

oButton = tkinter.Button(text='Click me', command=myFunction)
```

When you create a button with `tkinter`, you must pass in a function (or a method of an object), which will be called back when the user clicks the button. Here, we are passing `myFunction` as the function to be called back. (This call is using keyword parameters, which will be discussed at length in Chapter 7.) The `tkinter` button remembers that function as the callback, and when the user clicks the resulting button, it calls the function `myFunction()`.

You can also use a callback when you initiate some action that may take some time. Instead of waiting for the action to finish and causing the program appear to freeze for a period of time, you provide a callback to be called when the action is completed. For example, imagine that you want to make a request across the internet. Rather than making a call and waiting for that call to return data, which may take a long time, there are packages that allow you to use the approach of making the call and setting a callback. That way, the program can continue running, and the user is not locked

out of it. This often involves multiple Python threads and is beyond the scope of this book, but the technique of using a callback is the general way that it is done.

Using a Callback with SimpleButton

To demonstrate this concept, we'll make a minor modification to the `SimpleButton` class to allow it to accept a callback. As an additional optional parameter, the caller can provide a function or method of an object to be called back when a click on a `SimpleButton` object happens. Each instance of `SimpleButton` remembers the callback in an instance variable. When the user completes a click, the instance of `SimpleButton` calls the callback.

The main program in Listing 6-8 creates three instances of the `SimpleButton` class, each of which handles the button click in a different way. The first button, `oButtonA`, provides no callback; `oButtonB` provides a callback to a function; and `oButtonC` specifies a callback to a method of an object.

File: PygameDemo9_SimpleButtonWithCallback/Main_SimpleButtonCallback.py

```
#  pygame demo 9 - 3-button test with callbacks

# 1 - Import packages
import pygame
from pygame.locals import *
from SimpleButton import *
import sys

# #2 - Define constants
GRAY = (200, 200, 200)
WINDOW_WIDTH = 400
WINDOW_HEIGHT = 100
FRAMES_PER_SECOND = 30

# Define a function to be used as a "callback"
def myCallBackFunction(): ❶
    print('User pressed Button B, called myCallBackFunction')

# Define a class with a method to be used as a "callback"
class CallBackTest(): ❷
--- snipped any other methods in this class ---

    def myMethod(self):
        print('User pressed ButtonC, called myMethod of the CallBackTest object')

# 3 - Initialize the world
pygame.init()
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
clock = pygame.time.Clock()

# 4 - Load assets: image(s), sound(s), etc.

# 5 - Initialize variables
```

```
oCallBackTest = CallBackTest() ❸
# Create instances of SimpleButton
# No call back
oButtonA = SimpleButton(window, (25, 30), ❹
                        'images/buttonAUp.png',
                        'images/buttonADown.png')
# Specifying a function to call back
oButtonB = SimpleButton(window, (150, 30),
                        'images/buttonBUp.png',
                        'images/buttonBDown.png',
                        callBack=myCallBackFunction)
# Specifying a method of an object to call back
oButtonC = SimpleButton(window, (275, 30),
                        'images/buttonCUp.png',
                        'images/buttonCDown.png',
                        callBack=oCallBackTest.myMethod)
counter = 0

# 6 - Loop forever
while True:

    # 7 - Check for and handle events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        # Pass the event to the button, see if it has been clicked on
        if oButtonA.handleEvent(event): ❺
            print('User pressed button A, handled in the main loop')

        # oButtonB and oButtonC have callbacks,
        # no need to check result of these calls
        oButtonB.handleEvent(event) ❻

        oButtonC.handleEvent(event) ❼

    # 8 - Do any "per frame" actions
    counter = counter + 1

    # 9 - Clear the window
    window.fill(GRAY)

    # 10 - Draw all window elements
    oButtonA.draw()
    oButtonB.draw()
    oButtonC.draw()

    # 11 - Update the window
    pygame.display.update()

    # 12 - Slow things down a bit
    clock.tick(FRAMES_PER_SECOND)  # make pygame wait
```

Listing 6-8: A version of the main program that handles button clicks three different ways

We start with a simple function, `myCallBackFunction()` ❶, that just prints a message to announce that it has been called. Next, we have a `CallBackTest` class that contains the method `myMethod()` ❷, which prints its own message to announce that it's been called. We create an `oCallBackTest` object from the `CallBackTest` class ❸. We need this object so we can set up a callback to `oCallBack.myMethod()`.

Then we create three `SimpleButton` objects, each using a different approach ❹. The first, `oButtonA`, has no callback. The second, `oButtonB`, sets its callback to the function `myCallBackFunction()`. The third, `oButtonC`, sets its callback to `oCallBack.myMethod()`.

In the main loop, we check for the user clicking on any of the three buttons by calling the `handleEvent()` method of each button. Since `oButtonA` has no callback, we must check if the value returned is `True` ❺ and, if so, perform an action. When `oButtonB` is clicked ❻, the `myCallBackFunction()` function will be called and will print its message. When `oButtonC` is clicked ❼, the `myMethod()` method of the `oCallBackTest` object will be called and will print its message.

Some programmers prefer using a callback approach, because the target to be called is set up when you create the object. It's important to understand this technique, especially if you are using a package that requires it. However, I will use the original approach of checking for the value returned by a call to `handleEvent()` in all my demonstration code.

Summary

In this chapter, I showed how you can start with a procedural program and extract related code to build a class. We created a `Ball` class to demonstrate this, then modified the main code of our demo program from the previous chapter to call methods of the class to tell the `Ball` object *what* to do, without worrying about *how* it achieves the outcome. With all the related code in a separate class, it's easy to create a list of objects and instantiate and manage as many objects as we want to.

We then built a `SimpleButton` class and a `SimpleText` class that hide complexity inside their implementation and create highly reusable code. In the next chapter, I'll build on these classes to develop "professional-strength" button and text display classes.

Finally, I introduced the concept of a callback, where you pass in a function or method in a call to an object. The callback is later called back when an event happens or an action completes.

7

PYGAME GUI WIDGETS

Pygame allows programmers to take the text-based language of Python and use it to build GUI-based programs. Windows, pointing devices, clicking, dragging, and sounds have all become standard parts of our experience using computers. Unfortunately, the pygame package doesn't come with built-in basic user interface elements, so we need to build them ourselves. We'll do so with `pygwidgets`, a library of GUI widgets.

This chapter explains how standard widgets such as images, buttons, and input or output fields can be built as classes and how client code uses them. Building each element as a class allows programmers to incorporate multiple instances of each element when creating a GUI. Before we get started building these GUI widgets, however, I first need to discuss one more Python feature: passing data in a call to a function or method.

Passing Arguments into a Function or Method

The arguments in a call to a function and the parameters defined in the function have a one-to-one relationship, so that the value of the first argument is given to the first parameter, the value of the second argument is given to the second parameter, and so on.

Figure 7-1, duplicated from Chapter 3, shows that the same is true when you make a call to a method of an object. We can see that the first parameter, which is always `self`, is set to the object in the call.

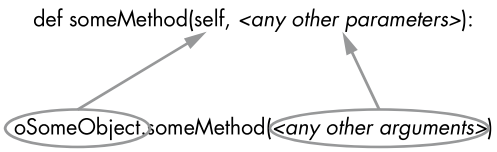

Figure 7-1: How arguments passed into a method match up with its parameters

However, Python (and some other languages) allows you to make some of the arguments optional. If an optional argument isn't provided in a call, we can provide a default value to use in the function or method instead. I'll explain by means of a real-world analogy.

If you order a hamburger at a Burger King restaurant, your burger will come with ketchup, mustard, and pickles. But Burger King is famous for saying, "You can have it your way." If you want some other combination of condiments, you must say what you want (or don't want) when you make your order.

We'll start by writing an `orderBurgers()` function that simulates making a burger order in the regular way we've been defining functions, without implementing default values:

```
def orderBurgers(nBurgers, ketchup, mustard, pickles):
```

You must specify the number of hamburgers you want to order, but ideally, if you want the defaults of `True` for adding ketchup, mustard, and pickles, you shouldn't need to pass in any more arguments. So, to order two hamburgers with the standard defaults, you might think your call should look like this:

```
orderBurgers(2)  # with ketchup, mustard, and pickles
```

However, in Python, this will trigger an error because there is a mismatch between the number of arguments in the call and the number of parameters specified in the function:

```
TypeError: orderBurgers() missing 3 required positional arguments: 'ketchup',
'mustard', and 'pickles'
```

Let's see how Python allows us to set up optional parameters that can be given default values if nothing is specified.

Positional and Keyword Parameters

Python has two different types of parameters: positional parameters and keyword parameters. *Positional parameters* are the type that we're already familiar with, where each argument in a call has a matching parameter in the function or method definition.

A *keyword parameter* allows you to specify a default value. You write a keyword parameter as a variable name, an equal sign, and a default value, like this:

```
def someFunction(<keywordParameter>=<default value>):
```

You can have multiple keyword parameters, each with a name and a default value.

A function or method can have both positional parameters and keyword parameters, in which case you must specify all positional parameters *before* any keyword parameters:

```
def someOtherFunction(positionalParam1, positionalParam2, ...
          <keywordParameter1>=<default value 1>,
          <keywordParameter2>=<default value 2>, ...):
```

Let's rewrite `orderBurgers()` to use one positional parameter and three keyword parameters with default values, like this:

```
def orderBurgers(nBurgers, ketchup=True, mustard=True, pickles=True):
```

When we make a call to this function, `nBurgers` is a positional parameter and therefore must be specified as an argument in every call. The other three are keyword parameters. If no values are passed for `ketchup`, `mustard`, and `pickles`, the function will use the default value of `True` for each of those parameter variables. Now we can order two burgers with all the condiments like this:

```
orderBurgers(2)
```

If we want something other than a default value, we can specify the name of the keyword parameter and a different value in our call. For example, if we only want ketchup on our two burgers, we can make the call this way:

```
orderBurgers(2, mustard=False, pickles=False)
```

When the function runs, the values of the `mustard` and `pickles` variables are set to `False`. Since we did not specify a value for `ketchup`, it is given the default of `True`.

You can also make the call specifying all arguments positionally, including those written as keyword parameters. Python will use the ordering of your arguments to assign each parameter the correct value:

```
orderBurgers(2, True, False, False)
```

In this call, we are again specifying two burgers with ketchup, no mustard, and no pickles.

Additional Notes on Keyword Parameters

Let's quickly go over a few conventions and tips for using keyword parameters. As a Python convention, when you use keyword parameters and keywords with arguments, the equal sign between the keyword and the value should *not* have spaces around it, to show that these are not typical assignment statements. These lines are properly formatted:

```
def orderBurgers(nBurgers, ketchup=True, mustard=True, pickles=True):

orderBurgers(2, mustard=False)
```

These lines will also work fine, but they don't follow the formatting convention and are less readable:

```
def orderBurgers(nBurgers, ketchup = True, mustard = True, pickles = True):

orderBurgers(2, mustard = False)
```

When calling a function that has both positional parameters and keyword parameters, you must provide values for all the positional parameters first, before any optional keyword parameters.

Keyword arguments in calls can be specified in any order. Calls to our `orderBurgers()` function could be made in various ways, such as:

```
orderBurgers(2, mustard=False, pickles=False)  # only ketchup
```

or:

```
orderBurgers(2, pickles=False, mustard=False, ketchup=False)  # plain
```

All keyword parameters will be given the appropriate values, independent of the order of the arguments.

While all the default values in the `orderBurgers()` example were Boolean values, a keyword parameter can have a default value of any data type. For example, we could write a function to allow a customer to make an ice cream order like this:

```
def orderIceCream(flavor, nScoops=1, coneOrCup='cone', sprinkles=False):
```

The caller must specify a flavor, but by default will get one scoop in a cone with no sprinkles. The caller could override these defaults with different keyword values.

Using None as a Default Value

It's sometimes helpful to know whether the caller passed in a value for a keyword parameter or not. For this example, the caller orders a pizza. At a minimum, the caller must specify a size. The second parameter will be a style that defaults to `'regular'` but could be `'deepdish'`. As a third parameter,

the caller can optionally pass in a single desired topping. If the caller wants a topping, we must charge them extra.

In Listing 7-1, we'll use a positional parameter for the `size` and keyword parameters for the `style` and `topping`. The default for `style` is the string `'regular'`. Since the topping choice is optional, we'll use the special Python value of `None` as the default, but the caller may pass in the topping of their choice.

File: OrderPizzaWithNone.py

```
def orderPizza(size, style='regular', topping=None):
    # Do some calculations based on the size and style
    # Check if a topping was specified
    PRICE_OF_TOPPING = 1.50  # price for any topping

    if size == 'small':
        price = 10.00
    elif size == 'medium':
        price = 14.00
    else: # large
        price = 18.00

    if style == 'deepdish':
        price = price + 2.00 # charge extra for deepdish

    line = 'You have ordered a ' + size + ' ' + style + ' pizza with '
❶   if topping is None:  # check if no topping was passed in
        print(line + 'no topping')
    else:
        print(line + topping)
        price = price + PRICE_OF_TOPPING

    print('The price is $', price)
    print()

# You could order a pizza in the following ways:
❷ orderPizza('large')   # large, defaults to regular, no topping

orderPizza('large', style='regular')  # same as above

❸ orderPizza('medium', style='deepdish', topping='mushrooms')

orderPizza('small', topping='mushrooms') # style defaults to regular
```

Listing 7-1: A function with a keyword parameter defaulting to `None`

The first and second calls would be seen as the same, with the value of the variable `topping` set to `None` ❷. In the third and fourth calls, the value of `topping` is set to `'mushrooms'` ❸. Because `'mushrooms'` is not `None`, in these calls the code would add in an extra charge for a topping on the pizzas ❶.

Using `None` as a default value for a keyword parameter gives you a way to see if the caller provided a value in the call. This may be a very subtle use of keyword parameters, but it will be very useful in our upcoming discussion.

Choosing Keywords and Default Values

Using default values makes calling functions and methods simpler, but there is a downside. Your choice of each keyword for keyword parameters is very important. Once programmers start making calls that override default values, it's very difficult to change the name of a keyword parameter because that name must be changed in *all* calls to the function or method in lockstep. Otherwise, code that was working will break. For more widely distributed code, this can potentially cause a great deal of pain to programmers using your code. Bottom line, don't change the name of a keyword parameter unless it is absolutely necessary. So, choose wisely!

It's also very important to use default values that should suit the widest possible range of users. (On a personal note, I *hate* mustard! Whenever I go to Burger King, I have to remember to specify no mustard or I'll get what I consider to be an inedible hamburger. I think they made a bad default choice.)

Default Values in GUI Widgets

In the next section, I'll present a collection of classes that you can use to easily create GUI elements such as buttons and text fields within pygame. These classes will each be initialized using a few positional parameters but will also have assorted optional keyword parameters, all with reasonable defaults to allow programmers to create GUI widgets by specifying only a few positional arguments. More precise control can be obtained by specifying values to overwrite the default values of keyword parameters.

For an in-depth example, we'll look at a widget to display text in the application's window. Text can be shown in a variety of fonts, font sizes, colors, background colors, and so on. We'll build a `DisplayText` class that will have default values for all of these attributes but will give client code the option of specifying different values.

The pygwidgets Package

The rest of this chapter will focus on the `pygwidgets` (pronounced "pig wijits") package, which was written with two goals in mind:

1. To demonstrate many different object-oriented programming techniques
2. To allow programmers to easily create and use GUI widgets in pygame programs

The `pygwidgets` package contains the following classes:

`TextButton`

Button built with standard art, using a text string

`CustomButton`

Button with custom artwork

`TextCheckBox`

Checkbox with standard art, built from a text string

`CustomCheckBox`

Checkbox with custom artwork

`TextRadioButton`

Radio buttons with standard art, built from a text string

`CustomRadioButton`

Radio buttons with custom artwork

`DisplayText`

Field used to display output text

`InputText`

Field where the user can type text

`Dragger`

Allows the user to drag an image

`Image`

Displays an image at a location

`ImageCollection`

Displays one of a collection of images at a location

`Animation`

Displays a sequence of images

`SpriteSheetAnimation`

Displays a sequence of images from a single larger image

Setting Up

To install `pygwidgets`, open the command line and enter the following:

```
python3 -m pip install -U pip --user
python3 -m pip install -U pygwidgets --user
```

These commands download and install the latest version of `pygwidgets` from the Python Package Index (PyPI). It is placed into a folder (named *site-packages*) that is available to all your Python programs. Once installed, you can use `pygwidgets` by including the following statement at the beginning of your programs:

```
import pygwidgets
```

This imports the entire package. After importing, you can instantiate objects from its classes and call the methods of those objects.

The most current documentation of pygwidgets is at *https://pygwidgets.readthedocs.io/en/latest/*. If you'd like to view the source code for the package, it's available via my GitHub repository at *https://github.com/IrvKalb/pygwidgets/*.

Overall Design Approach

As shown in Chapter 5, one of the first things you do in every pygame program is to define the window of the application. The following line creates an application window and saves a reference to it in a variable named window:

```
window = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
```

As we will soon see, whenever we instantiate any widget, we will need to pass in the window variable so the widget can draw itself in the application's window.

Most widgets in pygwidgets work in a similar way, typically involving these three steps:

1. Before the main while loop starts, create an instance of the widget, passing in any initialization arguments.
2. In the main loop, whenever any event happens, call the handleEvent() method of the widget (passing in the event object).
3. At the bottom of the main loop, call the draw() method of the widget.

Step 1 in using any widget is to instantiate one with a line like this:

```
oWidget = pygwidgets.<SomeWidgetClass>(window, loc, <other arguments as needed>)
```

The first argument is always the window of the application. The second argument is always the location in the window at which to display the widget, given as a tuple: (x, y).

Step 2 is to handle any event that could affect the widget by calling the object's handleEvent() method inside the event loop. If any event (like a mouse click or button press) happens and the widget handles the event, this call will return True. The code at the top of the main while loop generally looks like this:

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

        if oWidget.handleEvent(event):
            # The user has done something to oWidget that we should respond to
            # Add code here
```

Step 3 is to add a line near the bottom of the while loop to call the draw() method of the widget, to make it appear it in the window:

```
oWidget.draw()
```

Since we specified the window to draw into, the location, and any details that affect the appearance of the widget in step 1, we don't pass anything in the call to `draw()`.

Adding an Image

Our first example will be the simplest widget: we'll use the `Image` class to display an image in a window. When you instantiate an `Image` object, the only required arguments are the window, the location in the window to draw the image, and the path to the image file. Create the `Image` object before the main loop starts, like so:

```
oImage = pygwidgets.Image(window, (100, 200), 'images/SomeImage.png')
```

The path used here assumes that the project folder containing the main program also contains a folder named *images*, inside which is the *SomeImage.png* file. Then, in the main loop you just need to call the object's `draw()` method:

```
oImage.draw()
```

The `draw()` method of the `Image` class contains a call to `blit()` to actually draw the image, so you never need to call `blit()` directly. To move the image, you can call its `setLoc()` method (short for set location), specifying the new x- and y-coordinates as a tuple:

```
oImage.setLoc((newX, newY))
```

The next time the image is drawn, it will show up at the new coordinates. The documentation lists many additional methods that you can call to flip, rotate, scale, get the image's location and rectangle, and so on.

THE SPRITE MODULE

Pygame has a built-in module to show images in a window, called the *sprite module*. Such images are called *sprites*. The sprite module provides a `Sprite` class for handling individual sprites and a `Group` class for handling multiple `Sprite` objects. Together, these classes provide excellent functionality, and if you intend to do heavy-duty pygame programming, it is probably worth your time to look into them. However, in order to explain the underlying OOP concepts, I have chosen not to use those classes. Instead, I will proceed with general GUI elements so that they can be used in any environment and language. If you want to learn more about the sprite module, see the tutorial at *https://www.pygame.org/docs/tut/SpriteIntro.html*.

Adding Buttons, Checkboxes, and Radio Buttons

When you instantiate a button, checkbox, or radio button widget in `pygwidgets`, you have two options: instantiate a text version that draws its own art and adds a text label based on a string you pass in, or instantiate a custom version where you supply the art. Table 7-1 shows the different button classes that are available.

Table 7-1: Text and Custom Button Classes in `pygwidgets`

	Text version (builds art on the fly)	Custom version (uses your artwork)
Button	`TextButton`	`CustomButton`
Checkbox	`TextCheckBox`	`CustomCheckBox`
Radio button	`TextRadioButton`	`CustomRadioButton`

The differences between the text and custom versions of these classes are only relevant during instantiation. Once you create an object from a text or custom button class, all the remaining methods of the pair of classes are identical. To make this clear, let's take a look at the `TextButton` and `CustomButton` classes.

TextButtons

Here is the actual definition of the `__init__()` method of the `TextButton` class in `pygwidgets`:

```
def __init__(self, window, loc, text,
             width=None,
             height=40,
             textColor=PYGWIDGETS_BLACK,
             upColor=PYGWIDGETS_NORMAL_GRAY,
             overColor=PYGWIDGETS_OVER_GRAY,
             downColor=PYGWIDGETS_DOWN_GRAY,
             fontName=DEFAULT_FONT_NAME,
             fontSize=DEFAULT_FONT_SIZE,
             soundOnClick=None,
             enterToActivate=False,
             callback=None
             nickname=None):
```

However, rather than reading through the code of a class, a programmer will typically refer to its documentation. As mentioned earlier, you can find the complete documentation for `pygwidgets` at *https://pygwidgets.readthedocs.io/en/latest/*.

You can also view documentation of a class by calling the built-in `help()` function in the Python shell like so:

```
>>> help(pygwidgets.TextButton)
```

When you create an instance of a `TextButton`, you are only required to pass in the window, the location in the window, and the text to be shown on the button. If you only specify these positional parameters, your button will use reasonable defaults for the width and height, the background colors for the four states of the button (different shades of gray), the font, and the font size. By default, no sound effect will be played when the user clicks on the button.

The code to create a `TextButton` using all the defaults looks like this:

```
oButton = pygwidgets.TextButton(window, (50, 50), 'Text Button')
```

The code in the `__init__()` method of the `TextButton` class uses the pygame drawing methods to construct its own art for all four states (up, down, over, and disabled). The preceding line creates an "up" version of a button that looks like Figure 7-2.

Figure 7-2: A `TextButton` using defaults

You can override any or all of the default parameters with keyword values like so:

```
oButton = pygwidgets.TextButton(window, (50, 50), 'Text Button',
                                width=200,
                                height=30,
                                textColor=(255, 255, 128),
                                upColor=(128, 0, 0),
                                fontName='Courier',
                                fontSize=14,
                                soundOnClick='sounds/blip.wav',
                                enterToActivate=True)
```

This instantiation will create a button that looks like Figure 7-3.

Figure 7-3: A `TextButton` using keyword arguments for font, size, colors, and so on

The image-switching behavior of these two buttons would work exactly the same way; the only differences would be in the appearance of the images.

CustomButtons

The `CustomButton` class allows you to use your own art for a button. To instantiate a `CustomButton`, you need only pass in a window, a location, and a path to the image of the up state of the button. Here is an example:

```
restartButton = pygwidgets.CustomButton(window, (100, 430),
                                        'images/RestartButtonUp.png')
```

The down, over, and disabled states are optional keyword arguments, and for any of these where no value is passed in, CustomButton will use a copy of the up image. It's more typical (and strongly suggested) to pass in paths for the optional images, like so:

```
restartButton = pygwidgets.CustomButton(window, (100, 430),
                        'images/RestartButtonUp.png',
                         down='images/RestartButtonDown.png',
                         over='images/RestartButtonOver.png',
                         disabled='images/RestartButtonDisabled.png',
                         soundOnClick='sounds/blip.wav',
                         nickname='restart')
```

Here we also specified a sound effect that should be played when the user clicks the button, and we provided an internal nickname we can use later.

Using Buttons

After instantiation, here's some typical code to use a button object, oButton, independent of it being a TextButton or a CustomButton:

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

        if oButton.handleEvent(event):
            # User has clicked this button
            <Any code you want to run here when the button is clicked>
--- snip ---
oButton.draw()  # at the bottom of the while loop, tell it to draw
```

Every time we detect an event, we need to call the handleEvent() method of the button to allow it to react to the user's actions. This call normally returns False but will return True when the user completes a click on the button. At the bottom of the main while loop, we need to call the draw() method of the button to allow it to draw itself.

Text Output and Input

As we saw in Chapter 6, handling text input and output in pygame is tricky, but here I'll introduce new classes for a text display field and an input text field. Both of these classes have minimal required (positional) parameters, and they have reasonable defaults for other attributes (font, font size, color, and so on) that are easily overridden.

Text Output

The pygwidgets package contains a DisplayText class for showing text that is a more full-featured version of the SimpleText class from Chapter 6. When you

instantiate a DisplayText field, the only required arguments are the window and the location. The first keyword parameter is value, which may be specified with a string as starting text to be shown in the field. This is typically used for a default end user value or for text that never changes, like a label or instructions. Since value is the first keyword parameter, it can be given as either a positional or a keyword argument. For example, this:

```
oTextField = pygwidgets.DisplayText(window, (10, 400), 'Hello World')
```

will work the same way as this:

```
oTextField = pygwidgets.DisplayText(window, (10, 400), value='Hello World')
```

You can also customize the look of the output text by specifying any or all of the optional keyword parameters. For example:

```
oTextField = pygwidgets.DisplayText(window, (10, 400),
                                    value='Some title text',
                                    fontName='Courier',
                                    fontSize=40,
                                    width=150,
                                    justified='center',
                                    textColor=(255, 255, 0))
```

The DisplayText class has a number of additional methods, the most important of which is setValue(), which you call to change the text drawn in the field:

```
oTextField.setValue('Any new text you want to see')
```

At the bottom of the main while loop, you need to call the object's draw() method:

```
oTextField.draw()
```

And of course, you can create as many DisplayText objects as you wish, each displaying different text and each with its own font, size, color, and so on.

Text Input

In a typical text-based Python program, to get input from the user you would make a call to the input() function, which stops the program until the user enters text in the shell window. But in the world of event-driven GUI programs, the main loop never stops. Therefore, we must use a different approach.

For text input from the user, a GUI program typically presents a field that the user can type in. An input field must deal with all keyboard keys, some of which show while others are used for editing or cursor movement within the field. It must also allow for the user holding down a key to repeat it. The pygwidgets InputText class provides all this functionality.

The only required arguments to instantiate an `InputText` object are the window and a location:

```
oInputField = pygwidgets.InputText(window, (10, 100))
```

However, you can customize the text attributes of an `InputText` object by specifying optional keyword arguments:

```
oInputField = pygwidgets.InputText(window, (10, 400),
                                   value='Starting Text',
                                   fontName='Helvetica',
                                   fontSize=40,
                                   width=150,
                                   textColor=(255, 255, 0))
```

After instantiating an `InputText` field, the typical code in the main loop would look like this:

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

        if oInputField.handleEvent(event):
            # User has pressed Enter or Return
            userText = oInputField.getValue()  # get the text the user entered
            <Any code you want to run using the user's input>
--- snip ---
    oInputField.draw()  # at the bottom of the main while loop
```

For every event, we need to call the `handleEvent()` method of the `InputText` field to allow it to react to keystrokes and mouse clicks. This call normally returns `False`, but when the user presses ENTER or RETURN, it returns `True`. We can then retrieve the text that the user entered by calling the `getValue()` method of the object.

At the bottom of the main `while` loop, we need to call the `draw()` method to allow the field to draw itself.

If a window contains multiple input fields, key presses are handled by the field with current keyboard focus, which is changed when a user clicks in a different field. If you want to allow a field to have initial keyboard focus, then you can set the `initialFocus` keyword parameter to `True` in the `InputText` object of your choice when you create that object. Further, if you have multiple `InputText` fields in a window, a typical user interface design approach is to include an OK or Submit button. When this button is clicked, you could then call the `getValue()` method of each field.

NOTE *At the time of writing, the* `InputText` *class does not handle highlighting multiple characters by dragging the mouse. If this functionality is added in a later version, no change will be required to programs that use* `InputText` *because the code will be entirely within that class. Any new behavior will be supported automatically in all* `InputText` *objects.*

Other pygwidgets Classes

As you saw at the beginning of this section, `pygwidgets` contains a number of other classes.

The `ImageCollection` class allows you to show any single image from a collection of images. For example, suppose you have images of a character facing front, left, back, and right. To represent all the potential images, you can build a dictionary like this:

```
imageDict = {'front':'images/front.png', 'left':'images/left.png',
             'back':'images/back.png', 'right':'images/right.png'}
```

You can then create an `ImageCollection` object, specifying this dictionary and the key of the image you want to start with. To change to a different image, you call the `replace()` method and pass in a different key. Calling the `draw()` method at the bottom of the loop always shows the current image.

The `Dragger` class displays a single image but allows the user to drag the image anywhere in the window. You must call its `handleEvent()` method in the event loop. When the user finishes dragging, `handleEvent()` returns `True`, and you can call the `Dragger` object's `getMouseUpLoc()` method to get the location where the user released the mouse button.

The `Animation` and `SpriteSheetAnimation` classes handle building and showing an animation. Both require a set of images to iterate through. The `Animation` class gets the images from individual files, while the `SpriteSheetAnimation` class requires a single image with evenly spaced internal images. We'll explore these classes more fully in Chapter 14.

pygwidgets Example Program

Figure 7-4 shows a screenshot of a sample program that demonstrates objects instantiated from many of the classes in `pygwidgets`, including `Image`, `DisplayText`, `InputText`, `TextButton`, `CustomButton`, `TextRadioButton`, `CustomRadioButton`, `TextCheckBox`, `CustomCheckBox`, `ImageCollection`, and `Dragger`.

The source of this example program can be found in the *pygwidgets_test* folder in my GitHub repository, *https://github.com/IrvKalb/pygwidgets/*.

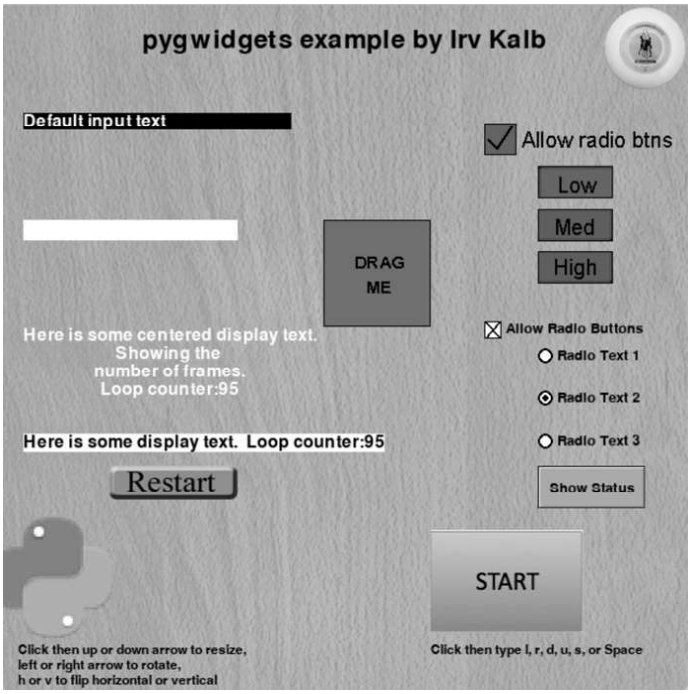

Figure 7-4: The window of a program that demonstrates objects instantiated from a variety of `pygwidgets` classes

The Importance of a Consistent API

One final note about building an API for a set of classes: whenever possible, it's a very good idea to build consistency into the parameters of methods in different, but similar, classes. As a good example, the first two parameters to the `__init__()` method of every class in `pygwidgets` are `window` and `loc`, in that order. If these had been in a different order in some calls, using the package as a whole would be much more difficult.

Additionally, if different classes implement the same functionality, it's a good idea to use the same method names. For example, many of the classes in `pygwidgets` have a method named `setValue()` and another named `getValue()`. I'll talk more about why this type of consistency is so important in the next two chapters.

Summary

This chapter provided an introduction to the object-oriented `pygwidgets` package of graphical user interface widgets. We began by discussing default values for parameters in methods, and I explained that a keyword parameter allows for a default value to be used if no matching argument value is specified in a call.

I then introduced you to the `pygwidgets` module, which contains a number of prebuilt GUI widget classes, and showed you how to use several of these. Finally, I showed a sample program that provides examples of most of these widgets.

There are two key advantages to writing classes like those in `pygwidgets`. First, classes can hide complexity in methods. Once you have your class working correctly, you never have to worry about the internal details again. Second, you can reuse the code by creating as many instances of a class as you need. Your classes can provide basic functionality by including keyword parameters with well-chosen default values. However, the default values can easily be overwritten to allow for customization.

You can publish the interfaces of your classes for other programmers (and yourself) to take advantage of in different projects. Good documentation and consistency go a long way toward making these types of classes highly usable.