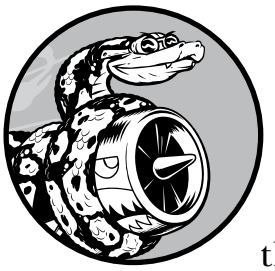
PROJECT 1

ALIEN INVASION

12

A SHIP THAT FIRES BULLETS



Let's build a game! We'll use Pygame, a collection of fun, powerful Python modules that manage graphics, animation, and even sound, making it easier for you to build sophisticated games. With Pygame handling tasks like drawing images to the screen, you can skip much of the tedious, difficult coding and focus on the higher-level logic of game dynamics.

In this chapter, you'll set up Pygame and then create a ship that moves right and left, and fires bullets in response to player input. In the next two chapters, you'll create a fleet of aliens to destroy, and then continue to make refinements, such as setting limits on the number of ships you can use and adding a scoreboard.

From this chapter you'll also learn to manage large projects that span multiple files. We'll refactor a lot of code and manage file contents to keep our project organized and the code efficient.

Making games is an ideal way to have fun while learning a language. It's deeply satisfying to watch others play a game you wrote, and writing a simple game will help you understand how professional games are written. As you work through this chapter, enter and run the code to understand how each block of code contributes to overall gameplay. Experiment with different values and settings to gain a better understanding of how to refine interactions in your own games.

NOTE *Alien Invasion will span a number of different files, so make a new folder on your system called* alien_invasion. *Be sure to save all files for the project to this folder so your* `import` *statements will work correctly.*

Planning Your Project

When building a large project, it's important to prepare a plan before you begin to write your code. Your plan will keep you focused and make it more likely that you'll complete the project.

Let's write a description of the overall gameplay. Although this description doesn't cover every detail of Alien Invasion, it provides a clear idea of how to start building the game:

> In Alien Invasion, the player controls a ship that appears at the bottom center of the screen. The player can move the ship right and left using the arrow keys and shoot bullets using the spacebar. When the game begins, a fleet of aliens fills the sky and moves across and down the screen. The player shoots and destroys the aliens. If the player shoots all the aliens, a new fleet appears that moves faster than the previous fleet. If any alien hits the player's ship or reaches the bottom of the screen, the player loses a ship. If the player loses three ships, the game ends.

For the first phase of development, we'll make a ship that can move right and left. The ship should be able to fire bullets when the player presses the spacebar. After setting up this behavior, we can turn our attention to the aliens and refine the gameplay.

Installing Pygame

Before you begin coding, install Pygame. Here's how to do so on Linux, OS X, and Microsoft Windows.

If you're using Python 3 on Linux or if you're using OS X, you'll need to use pip to install Pygame. pip is a program that handles the downloading and installing of Python packages for you. The following sections will show you how to install packages with pip.

If you're using Python 2.7 on Linux or if you're using Windows, you won't need pip to install Pygame. Instead, move on to "Installing Pygame on Linux" on page 238 or "Installing Pygame on Windows" on page 240.

NOTE *Instructions for installing pip on all systems are included in the sections that follow because you'll need pip for the data visualization and web application projects. These instructions are also included in the online resources at* https://www.nostarch.com/pythoncrashcourse/. *If you have trouble with the instructions here, see if the online instructions work for you.*

Installing Python Packages with pip

The most recent versions of Python come with pip installed, so first check whether pip is already on your system. With Python 3, pip is sometimes called *pip3*.

Checking for pip on Linux and OS X

Open a terminal window and enter the following command:

```
$ pip --version
❶ pip 7.0.3 from /usr/local/lib/python3.5/dist-packages (python 3.5)
$
```

If you have only one version of Python installed on your system and you see output similar to this, move on to either "Installing Pygame on Linux" on page 238 or "Installing Pygame on OS X" on page 239. If you get an error message, try using `pip3` instead of `pip`. If neither version is installed on your system, go to "Installing pip" on page 238.

If you have more than one version of Python on your system, verify that pip is associated with the version of Python you're using—for example, `python 3.5` at ❶. If pip is associated with the correct version of Python, move on to "Installing Pygame on Linux" on page 238 or "Installing Pygame on OS X" on page 239. If pip is associated with the wrong version of Python, try using `pip3` instead of `pip`. If neither command works for the version of Python you're using, go to "Installing pip" on page 238.

Checking for pip on Windows

Open a terminal window and enter the following command:

```
$ python -m pip --version
❶ pip 7.0.3 from C:\Python35\lib\site-packages (python 3.5)
$
```

If your system has only one version of Python installed and you see output similar to this, move on to "Installing Pygame on Windows" on page 240. If you get an error message, try using `pip3` instead of `pip`. If neither version is installed on your system, move on to "Installing pip" on page 238.

If your system has more than one version of Python installed, verify that pip is associated with the version of Python you're using—for example, `python 3.5` at ❶. If pip is associated with the correct version of Python, move on to "Installing Pygame on Windows" on page 240. If pip is associated with

the wrong version of Python, try using `pip3` instead of `pip`. If neither command works for the version of Python you're using, move on to "Installing pip" next.

Installing pip

To install pip, go to *https://bootstrap.pypa.io/get-pip.py*. Save the file if prompted to do so. If the code for *get-pip.py* appears in your browser, copy and paste the program into your text editor and save the file as *get-pip.py*. Once *get-pip.py* is saved on your computer, you'll need to run it with administrative privileges because pip will be installing new packages to your system.

NOTE *If you can't find* get-pip.py, *go to* https://pip.pypa.io/, *click* ***Installation*** *in the left panel, and then under "Install pip," follow the link to* get-pip.py.

Installing pip on Linux and OS X

Use the following command to run *get-pip.py* with administrative privileges:

```
$ sudo python get-pip.py
```

NOTE *If you use the command* `python3` *to start a terminal session, you should use* ***`sudo python3 get-pip.py`*** *here.*

After the program runs, use the command **`pip --version`** (or **`pip3 --version`**) to make sure pip was installed correctly.

Installing pip on Windows

Use the following command to run *get-pip.py*:

```
$ python get-pip.py
```

If you use a different command to run Python in a terminal, make sure you use that command to run *get-pip.py*. For example, your command might be `python3 get-pip.py` or `C:\Python35\python get-pip.py`.

After the program runs, run the command **`python -m pip --version`** to make sure pip was installed successfully.

Installing Pygame on Linux

If you're using Python 2.7, install Pygame using the package manager. Open a terminal window and run the following command, which will download and install Pygame onto your system:

```
$ sudo apt-get install python-pygame
```

Test your installation in a terminal session by entering the following:

```
$ python
>>> import pygame
>>>
```

If no output appears, Python has imported Pygame and you're ready to move on to "Starting the Game Project" on page 240.

If you're running Python 3, two steps are required: installing the libraries Pygame depends on, and downloading and installing Pygame.

Enter the following to install the libraries Pygame needs. (If you use a command such as `python3.5` on your system, replace `python3-dev` with `python3.5-dev`.)

```
$ sudo apt-get install python3-dev mercurial
$ sudo apt-get install libsdl-image1.2-dev libsdl2-dev libsdl-ttf2.0-dev
```

This will install the libraries needed to run Alien Invasion successfully. If you want to enable some more advanced functionality in Pygame, such as the ability to add sounds, you can also add the following libraries:

```
$ sudo apt-get install libsdl-mixer1.2-dev libportmidi-dev
$ sudo apt-get install libswscale-dev libsmpeg-dev libavformat-dev libavcode-dev
$ sudo apt-get install python-numpy
```

Now install Pygame by entering the following (use `pip3` if that's appropriate for your system):

```
$ pip install --user hg+http://bitbucket.org/pygame/pygame
```

The output will pause for a moment after informing you which libraries Pygame found. Press ENTER, even though some libraries are missing. You should see a message stating that Pygame installed successfully.

To confirm the installation, run a Python terminal session and try to import Pygame by entering the following:

```
$ python3
>>> import pygame
>>>
```

If this works, move on to "Starting the Game Project" on page 240.

Installing Pygame on OS X

You'll need Homebrew to install some packages that Pygame depends on. If you haven't already installed Homebrew, see Appendix A for instructions.

To install the libraries that Pygame depends on, enter the following:

```
$ brew install hg sdl sdl_image sdl_ttf
```

This will install the libraries needed to run Alien Invasion. You should see output scroll by as each library is installed.

If you also want to enable more advanced functionality, such as including sound in games, you can install two additional libraries:

```
$ brew install sdl_mixer portmidi
```

Use the following command to install Pygame (use `pip` rather than `pip3` if you're running Python 2.7):

```
$ pip3 install --user hg+http://bitbucket.org/pygame/pygame
```

Start a Python terminal session and import Pygame to check whether the installation was successful (enter `python` rather than `python3` if you're running Python 2.7):

```
$ python3
>>> import pygame
>>>
```

If the `import` statement works, move on to "Starting the Game Project" below.

Installing Pygame on Windows

The Pygame project is hosted on a code-sharing site called Bitbucket. To install Pygame on your version of Windows, find a Windows installer at *https://bitbucket.org/pygame/pygame/downloads/* that matches the version of Python you're running. If you don't see an appropriate installer listed at Bitbucket, check *http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame*.

After you've downloaded the appropriate file, run the installer if it's a *.exe* file.

If you have a file ending in *.whl*, copy the file to your project directory. Open a command window, navigate to the folder that you copied the installer to, and use pip to run the installer:

```
> python -m pip install --user pygame-1.9.2a0-cp35-none-win32.whl
```

Starting the Game Project

Now we'll start building our game by first creating an empty Pygame window to which we can later draw our game elements, such as the ship and the aliens. We'll also have our game respond to user input, set the background color, and load a ship image.

Creating a Pygame Window and Responding to User Input

First, we'll create an empty Pygame window. Here's the basic structure of a game written in Pygame:

alien_invasion.py

```
import sys

import pygame

def run_game():
    # Initialize game and create a screen object.
❶   pygame.init()
❷   screen = pygame.display.set_mode((1200, 800))
    pygame.display.set_caption("Alien Invasion")

    # Start the main loop for the game.
❸   while True:

        # Watch for keyboard and mouse events.
❹       for event in pygame.event.get():
❺           if event.type == pygame.QUIT:
                sys.exit()

        # Make the most recently drawn screen visible.
❻       pygame.display.flip()

run_game()
```

First, we import the `sys` and `pygame` modules. The `pygame` module contains the functionality needed to make a game. We'll use the `sys` module to exit the game when the player quits.

Alien Invasion starts as the function `run_game()`. The line `pygame.init()` at ❶ initializes background settings that Pygame needs to work properly. At ❷, we call `pygame.display.set_mode()` to create a display window called `screen`, on which we'll draw all of the game's graphical elements. The argument `(1200, 800)` is a tuple that defines the dimensions of the game window. By passing these dimensions to `pygame.display.set_mode()`, we create a game window 1200 pixels wide by 800 pixels high. (You can adjust these values depending on the size of your display.)

The `screen` object is called a surface. A *surface* in Pygame is a part of the screen where you display a game element. Each element in the game, like the aliens or the ship, is a surface. The surface returned by `display.set_mode()` represents the entire game window. When we activate the game's animation loop, this surface is automatically redrawn on every pass through the loop.

The game is controlled by a `while` loop ❸ that contains an event loop and code that manages screen updates. An *event* is an action that the user performs while playing the game, such as pressing a key or moving the mouse. To make our program respond to events, we'll write an *event loop* to *listen* for an event and perform an appropriate task depending on the kind of event that occurred. The `for` loop at ❹ is an event loop.

To access the events detected by Pygame, we'll use the `pygame.event.get()` method. Any keyboard or mouse event will cause the `for` loop to run. Inside the loop, we'll write a series of `if` statements to detect and respond to specific events. For example, when the player clicks the game window's close button, a `pygame.QUIT` event is detected and we call `sys.exit()` to exit the game ❺.

The call to `pygame.display.flip()` at ❻ tells Pygame to make the most recently drawn screen visible. In this case it draws an empty screen each time through the `while` loop to erase the old screen so that only the new screen is visible. When we move the game elements around, `pygame.display.flip()` will continually update the display to show the new positions of elements and hide the old ones, creating the illusion of smooth movement.

The last line in this basic game structure calls `run_game()`, which initializes the game and starts the main loop.

Run this code now, and you should see an empty Pygame window.

Setting the Background Color

Pygame creates a black screen by default, but that's boring. Let's set a different background color:

alien_invasion.py

```
--snip--
def run_game():
    --snip--
    pygame.display.set_caption("Alien Invasion")

    # Set the background color.
❶   bg_color = (230, 230, 230)

    # Start the main loop for the game.
    while True:

        # Watch for keyboard and mouse events.
        --snip--

        # Redraw the screen during each pass through the loop.
❷       screen.fill(bg_color)

        # Make the most recently drawn screen visible.
        pygame.display.flip()

run_game()
```

First, we create a background color and store it in `bg_color` ❶. This color needs to be specified only once, so we define its value before entering the main `while` loop.

Colors in Pygame are specified as RGB colors: a mix of red, green, and blue. Each color value can range from 0 to 255. The color value (255, 0, 0) is red, (0, 255, 0) is green, and (0, 0, 255) is blue. You can mix RGB values to create 16 million colors. The color value (230, 230, 230) mixes equal amounts of red, blue, and green, which produces a light gray background color.

At ❷, we fill the screen with the background color using the `screen.fill()` method, which takes only one argument: a color.

Creating a Settings Class

Each time we introduce new functionality into our game, we'll typically introduce some new settings as well. Instead of adding settings throughout the code, let's write a module called `settings` that contains a class called `Settings` to store all the settings in one place. This approach allows us to pass around one settings object instead of many individual settings. In addition, it makes our function calls simpler and makes it easier to modify the game's appearance as our project grows. To modify the game, we'll simply change some values in *settings.py* instead of searching for different settings throughout our files.

Here's the initial `Settings` class:

settings.py

```
class Settings():
    """A class to store all settings for Alien Invasion."""

    def __init__(self):
        """Initialize the game's settings."""
        # Screen settings
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)
```

To make an instance of `Settings` and use it to access our settings, modify *alien_invasion.py* as follows:

alien_invasion.py

```
--snip--
import pygame

from settings import Settings

def run_game():
    # Initialize pygame, settings, and screen object.
    pygame.init()
❶   ai_settings = Settings()
❷   screen = pygame.display.set_mode(
        (ai_settings.screen_width, ai_settings.screen_height))
    pygame.display.set_caption("Alien Invasion")

    # Start the main loop for the game.
    while True:
        --snip--
        # Redraw the screen during each pass through the loop.
❸       screen.fill(ai_settings.bg_color)

        # Make the most recently drawn screen visible.
        pygame.display.flip()

run_game()
```

We import `Settings` into the main program file, and then create an instance of `Settings` and store it in `ai_settings` after making the call to `pygame.init()` ❶. When we create a screen ❷, we use the `screen_width` and `screen_height` attributes of `ai_settings`, and then we use `ai_settings` to access the background color when filling the screen at ❸ as well.

Adding the Ship Image

Now let's add the ship to our game. To draw the player's ship on screen, we'll load an image and then use the Pygame method `blit()` to draw the image.

When choosing artwork for your games, be sure to pay attention to licensing. The safest and cheapest way to start is to use freely licensed graphics that you can modify from a website like *http://pixabay.com/*.

You can use almost any type of image file in your game, but it's easiest if you use a bitmap (*.bmp*) file because Pygame loads bitmaps by default. Although you can configure Pygame to use other file types, some file types depend on certain image libraries that must be installed on your computer. (Most images you'll find are in *.jpg*, *.png*, or *.gif* formats, but you can convert them to bitmaps using tools like Photoshop, GIMP, and Paint.)

Pay particular attention to the background color in your chosen image. Try to find a file with a transparent background that you can replace with any background color using an image editor. Your games will look best if the image's background color matches your game's background color. Alternatively, you can match your game's background to the image's background.

For Alien Invasion, you can use the file *ship.bmp* (Figure 12-1), which is available in the book's resources through *https://www.nostarch.com/pythoncrashcourse/*. The file's background color matches the settings we're using in this project. Make a folder called *images* inside your main project folder (*alien_invasion*). Save the file *ship.bmp* in the *images* folder.

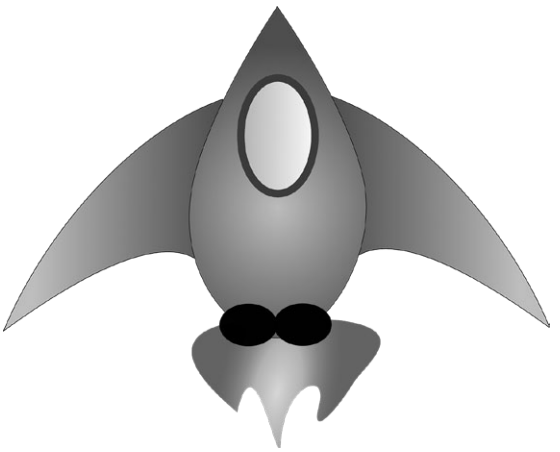


Figure 12-1: The ship for Alien Invasion

Creating the Ship Class

After choosing an image for the ship, we need to display it onscreen. To use our ship, we'll write a module called `ship`, which contains the class `Ship`. This class will manage most of the behavior of the player's ship.

ship.py

```
import pygame

class Ship():

    def __init__(self, screen):
        """Initialize the ship and set its starting position."""
        self.screen = screen

        # Load the ship image and get its rect.
❶       self.image = pygame.image.load('images/ship.bmp')
❷       self.rect = self.image.get_rect()
❸       self.screen_rect = screen.get_rect()

        # Start each new ship at the bottom center of the screen.
❹       self.rect.centerx = self.screen_rect.centerx
        self.rect.bottom = self.screen_rect.bottom

❺   def blitme(self):
        """Draw the ship at its current location."""
        self.screen.blit(self.image, self.rect)
```

First, we import the `pygame` module. The `__init__()` method of `Ship` takes two parameters: the `self` reference and the `screen` where we'll draw the ship. To load the image, we call `pygame.image.load()` ❶. This function returns a surface representing the ship, which we store in `self.image`.

Once the image is loaded, we use `get_rect()` to access the surface's `rect` attribute ❷. One reason Pygame is so efficient is that it lets you treat game elements like rectangles (*rects*), even if they're not exactly shaped like rectangles. Treating an element as a rectangle is efficient because rectangles are simple geometric shapes. This approach usually works well enough that no one playing the game will notice that we're not working with the exact shape of each game element.

When working with a `rect` object, you can use the x- and y-coordinates of the top, bottom, left, and right edges of the rectangle, as well as the center. You can set any of these values to determine the current position of the rect.

When you're centering a game element, work with the `center`, `centerx`, or `centery` attributes of a rect. When you're working at an edge of the screen, work with the `top`, `bottom`, `left`, or `right` attributes. When you're adjusting the horizontal or vertical placement of the rect, you can just use the `x` and `y` attributes, which are the x- and y-coordinates of its top-left corner. These attributes spare you from having to do calculations that game developers formerly had to do manually, and you'll find you'll use them often.

NOTE *In Pygame, the origin (0, 0) is at the top-left corner of the screen, and coordinates increase as you go down and to the right. On a 1200 by 800 screen, the origin is at the top-left corner, and the bottom-right corner has the coordinates (1200, 800).*

We'll position the ship at the bottom center of the screen. To do so, first store the screen's rect in `self.screen_rect` ❸, and then make the value of `self.rect.centerx` (the x-coordinate of the ship's center) match the `centerx` attribute of the screen's rect ❹. Make the value of `self.rect.bottom` (the y-coordinate of the ship's bottom) equal to the value of the screen rect's `bottom` attribute. Pygame will use these `rect` attributes to position the ship image so it's centered horizontally and aligned with the bottom of the screen.

At ❺ we define the `blitme()` method, which will draw the image to the screen at the position specified by `self.rect`.

Drawing the Ship to the Screen

Now let's update *alien_invasion.py* so it creates a ship and calls the ship's `blitme()` method:

alien_invasion.py

```
--snip--
from settings import Settings
from ship import Ship

def run_game():
    --snip--
    pygame.display.set_caption("Alien Invasion")

    # Make a ship.
❶   ship = Ship(screen)

    # Start the main loop for the game.
    while True:
        --snip--
        # Redraw the screen during each pass through the loop.
        screen.fill(ai_settings.bg_color)
❷       ship.blitme()

        # Make the most recently drawn screen visible.
        pygame.display.flip()

run_game()
```

We import `Ship` and then make an instance of `Ship` (named `ship`) after the screen has been created. It must come before the main `while` loop ❶ so we don't make a new instance of the ship on each pass through the loop. We draw the ship onscreen by calling `ship.blitme()` after filling the background, so the ship appears on top of the background ❷.

When you run *alien_invasion.py* now, you should see an empty game screen with our rocket ship sitting at the bottom center, as shown in Figure 12-2.

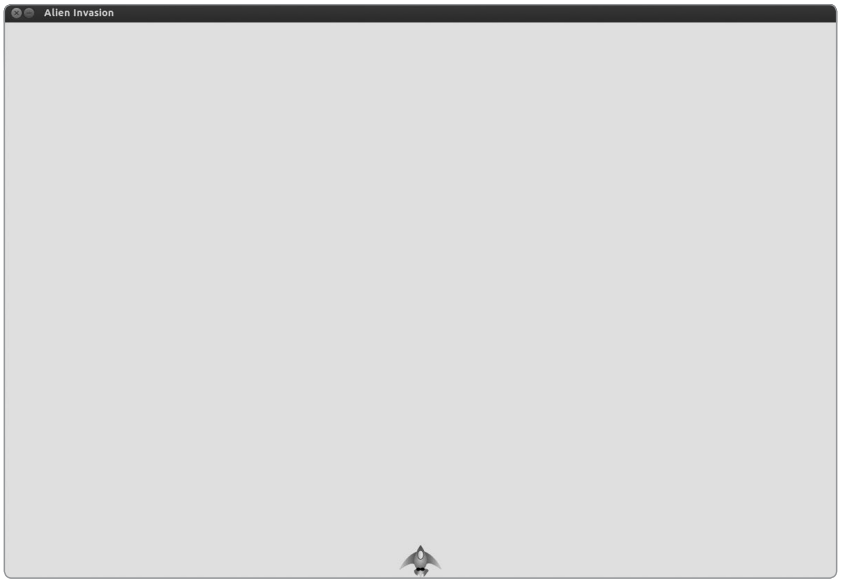


Figure 12-2: Alien Invasion with the ship at the bottom center of the screen

Refactoring: the game_functions Module

In larger projects, you'll often refactor code you've written before adding more code. *Refactoring* simplifies the structure of the code you've already written, making it easier to build on. In this section we'll create a new module called `game_functions`, which will store a number of functions that make Alien Invasion work. The `game_functions` module will prevent *alien_invasion.py* from becoming too lengthy and will make the logic in *alien_invasion.py* easier to follow.

The check_events() Function

We'll start by moving the code that manages events to a separate function called `check_events()`. This will simplify `run_game()` and isolate the event management loop. Isolating the event loop allows you to manage events separately from other aspects of the game, like updating the screen.

Place `check_events()` in a separate module called `game_functions`:

game_functions.py

```
import sys

import pygame

def check_events():
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

This module imports sys and pygame, which are used in the event checking loop. The function needs no parameters at this point, and the body is copied from the event loop in *alien_invasion.py*.

Now let's modify *alien_invasion.py* so it imports the game_functions module, and we'll replace the event loop with a call to check_events():

alien_invasion.py

```
import pygame

from settings import Settings
from ship import Ship
import game_functions as gf

def run_game():
    --snip--
    # Start the main loop for the game.
    while True:
        gf.check_events()

        # Redraw the screen during each pass through the loop.
        --snip--
```

We no longer need to import sys directly into the main program file, because it's only being used in the game_functions module now. We give the imported game_functions module the alias gf for simplification.

The update_screen() Function

Let's move the code for updating the screen to a separate function called update_screen() in *game_functions.py* to further simplify run_game():

game_functions.py

```
--snip--

def check_events():
    --snip--

def update_screen(ai_settings, screen, ship):
    """Update images on the screen and flip to the new screen."""
    # Redraw the screen during each pass through the loop.
    screen.fill(ai_settings.bg_color)
    ship.blitme()

    # Make the most recently drawn screen visible.
    pygame.display.flip()
```

The new update_screen() function takes three parameters: ai_settings, screen, and ship. Now we need to update the while loop from *alien_invasion.py* with a call to update_screen():

alien_invasion.py

```
--snip--
    # Start the main loop for the game.
```

```
    while True:
        gf.check_events()
        gf.update_screen(ai_settings, screen, ship)

run_game()
```

These two functions make the `while` loop simpler and will make further development easier. Instead of working inside `run_game()`, we can do most of our work in the module `game_functions`.

Because we wanted to start out working with code in a single file, we didn't introduce the `game_functions` module right away. This approach gives you an idea of a realistic development process: you start out writing your code as simply as possible, and refactor it as your project becomes more complex.

Now that our code is restructured to make it easier to add to, we can work on the dynamic aspects of the game!

TRY IT YOURSELF

12-1. Blue Sky: Make a Pygame window with a blue background.

12-2. Game Character: Find a bitmap image of a game character you like or convert an image to a bitmap. Make a class that draws the character at the center of the screen and match the background color of the image to the background color of the screen, or vice versa.

Piloting the Ship

Let's give the player the ability to move the ship right and left. To do this, we'll write code that responds when the player presses the right or left arrow key. We'll focus on movement to the right first, and then we'll apply the same principles to control movement to the left. As you do this, you'll learn how to control the movement of images on the screen.

Responding to a Keypress

Whenever the player presses a key, that keypress is registered in Pygame as an event. Each event is picked up by the `pygame.event.get()` method, so we need to specify in our `check_events()` function what kind of events to check for. Each keypress is registered as a `KEYDOWN` event.

When a `KEYDOWN` event is detected, we need to check whether the key that was pressed is one that triggers a certain event. For example, if the

right arrow key is pressed, we increase the ship's `rect.centerx` value to move the ship to the right:

game_functions.py

```
def check_events(ship):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

❶       elif event.type == pygame.KEYDOWN:
❷           if event.key == pygame.K_RIGHT:
                # Move the ship to the right.
❸               ship.rect.centerx += 1
```

We give the `check_events()` function a `ship` parameter, because the ship needs to move to the right when the right arrow key is pressed. Inside `check_events()` we add an `elif` block to the event loop to respond when Pygame detects a `KEYDOWN` event ❶. We check if the key pressed is the right arrow key (`pygame.K_RIGHT`) by reading the `event.key` attribute ❷. If the right arrow key was pressed, we move the ship to the right by increasing the value of `ship.rect.centerx` by 1 ❸.

We need to update the call to `check_events()` in *alien_invasion.py* so it passes `ship` as an argument:

alien_invasion.py

```
# Start the main loop for the game.
while True:
    gf.check_events(ship)
    gf.update_screen(ai_settings, screen, ship)
```

If you run *alien_invasion.py* now, you should see the ship move to the right one pixel every time you press the right arrow key. That's a start, but it's not an efficient way to control the ship. Let's improve this control by allowing continuous movement.

Allowing Continuous Movement

When the player holds down the right arrow key, we want the ship to continue moving right until the player releases the key. We'll have our game detect a `pygame.KEYUP` event so we'll know when the right arrow key is released; then we'll use the `KEYDOWN` and `KEYUP` events together with a flag called `moving_right` to implement continuous motion.

When the ship is motionless, the `moving_right` flag will be `False`. When the right arrow key is pressed, we'll set the flag to `True`, and when it's released, we'll set the flag to `False` again.

The `Ship` class controls all attributes of the ship, so we'll give it an attribute called `moving_right` and an `update()` method to check the status of the `moving_right` flag. The `update()` method will change the position of the ship if the flag is set to `True`. We'll call this method any time we want to update the position of the ship.

Here are the changes to the `Ship` class:

ship.py

```
class Ship():

    def __init__(self, screen):
        --snip--
        # Start each new ship at the bottom center of the screen.
        self.rect.centerx = self.screen_rect.centerx
        self.rect.bottom = self.screen_rect.bottom

        # Movement flag
❶       self.moving_right = False

❷   def update(self):
        """Update the ship's position based on the movement flag."""
        if self.moving_right:
            self.rect.centerx += 1

    def blitme(self):
        --snip--
```

We add a `self.moving_right` attribute in the `__init__()` method and set it to `False` initially ❶. Then we add `update()`, which moves the ship right if the flag is `True` ❷.

Now modify `check_events()` so that `moving_right` is set to `True` when the right arrow key is pressed and `False` when the key is released:

game_functions.py

```
def check_events(ship):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        --snip--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
❶               ship.moving_right = True

❷       elif event.type == pygame.KEYUP:
            if event.key == pygame.K_RIGHT:
                ship.moving_right = False
```

At ❶, we modify how the game responds when the player presses the right arrow key: instead of changing the ship's position directly, we merely set `moving_right` to `True`. At ❷, we add a new `elif` block, which responds to `KEYUP` events. When the player releases the right arrow key (`K_RIGHT`), we set `moving_right` to `False`.

Finally, we modify the `while` loop in *alien_invasion.py* so it calls the ship's `update()` method on each pass through the loop:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ship)
        ship.update()
        gf.update_screen(ai_settings, screen, ship)
```

The ship's position will update after we've checked for keyboard events and before we update the screen. This allows the ship's position to be updated in response to player input and ensures the updated position is used when drawing the ship to the screen.

When you run *alien_invasion.py* and hold down the right arrow key, the ship should move continuously to the right until you release the key.

Moving Both Left and Right

Now that the ship can move continuously to the right, adding movement to the left is easy. We'll again modify the `Ship` class and the `check_events()` function. Here are the relevant changes to `__init__()` and `update()` in `Ship`:

ship.py

```
    def __init__(self, screen):
        --snip--
        # Movement flags
        self.moving_right = False
        self.moving_left = False

    def update(self):
        """Update the ship's position based on movement flags."""
        if self.moving_right:
            self.rect.centerx += 1
        if self.moving_left:
            self.rect.centerx -= 1
```

In `__init__()`, we add a `self.moving_left` flag. In `update()`, we use two separate `if` blocks rather than an `elif` in `update()` to allow the ship's `rect.centerx` value to be increased and then decreased if both arrow keys are held down. This results in the ship standing still. If we used `elif` for motion to the left, the right arrow key would always have priority. Doing it this way makes the movements more accurate when switching from left to right, when the player might momentarily hold down both keys.

We have to make two adjustments to `check_events()`:

game_functions.py

```
def check_events(ship):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        --snip--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                ship.moving_right = True
            elif event.key == pygame.K_LEFT:
                ship.moving_left = True

        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_RIGHT:
                ship.moving_right = False
            elif event.key == pygame.K_LEFT:
                ship.moving_left = False
```

If a KEYDOWN event occurs for the K_LEFT key, we set moving_left to True. If a KEYUP event occurs for the K_LEFT key, we set moving_left to False. We can use elif blocks here because each event is connected to only one key. If the player presses both keys at once, two separate events will be detected.

If you run *alien_invasion.py* now, you should be able to move the ship continuously to the right and left. If you hold down both keys, the ship should stop moving.

Next, we'll further refine the movement of the ship. Let's adjust the ship's speed and limit how far the ship can move so it doesn't disappear off the sides of the screen.

Adjusting the Ship's Speed

Currently, the ship moves one pixel per cycle through the while loop, but we can take finer control of the ship's speed by adding a ship_speed_factor attribute to the Settings class. We'll use this attribute to determine how far to move the ship on each pass through the loop. Here's the new attribute in *settings.py*:

settings.py

```
class Settings():
    """A class to store all settings for Alien Invasion."""

    def __init__(self):
        --snip--

        # Ship settings
        self.ship_speed_factor = 1.5
```

We set the initial value of ship_speed_factor to 1.5. When we want to move the ship, we'll adjust its position by 1.5 pixels rather than 1 pixel.

We're using decimal values for the speed setting to give us finer control of the ship's speed when we increase the tempo of the game later on. However, rect attributes such as centerx store only integer values, so we need to make some modifications to Ship:

ship.py

```
class Ship():

❶   def __init__(self, ai_settings, screen):
        """Initialize the ship and set its starting position."""
        self.screen = screen
❷       self.ai_settings = ai_settings
        --snip--

        # Start each new ship at the bottom center of the screen.
        --snip--

        # Store a decimal value for the ship's center.
❸       self.center = float(self.rect.centerx)

        # Movement flags
        self.moving_right = False
        self.moving_left = False
```

```
    def update(self):
        """Update the ship's position based on movement flags."""
        # Update the ship's center value, not the rect.
        if self.moving_right:
❹           self.center += self.ai_settings.ship_speed_factor
        if self.moving_left:
            self.center -= self.ai_settings.ship_speed_factor

        # Update rect object from self.center.
❺       self.rect.centerx = self.center

    def blitme(self):
        --snip--
```

At ❶, we add `ai_settings` to the list of parameters for `__init__()`, so the ship will have access to its speed setting. We then turn the `ai_settings` parameter into an attribute, so we can use it in `update()` ❷. Now that we're adjusting the position of the ship by fractions of a pixel, we need to store the position in a variable that can store a decimal value. You can use a decimal value to set a `rect`'s attribute, but the `rect` will store only the integer portion of that value. To store the ship's position accurately, we define a new attribute `self.center`, which can hold decimal values ❸. We use the `float()` function to convert the value of `self.rect.centerx` to a decimal and store this value in `self.center`.

Now when we change the ship's position in `update()`, the value of `self.center` is adjusted by the amount stored in `ai_settings.ship_speed_factor` ❹. After `self.center` has been updated, we use the new value to update `self.rect.centerx`, which controls the position of the ship ❺. Only the integer portion of `self.center` will be stored in `self.rect.centerx`, but that's fine for displaying the ship.

We need to pass `ai_settings` as an argument when we create an instance of `Ship` in *alien_invasion.py*:

alien_invasion.py

```
--snip--
def run_game():
    --snip--
    # Make a ship.
    ship = Ship(ai_settings, screen)
    --snip--
```

Now any value of `ship_speed_factor` greater than one will make the ship move faster. This will be helpful in making the ship respond quickly enough to shoot down aliens, and it will let us change the tempo of the game as the player progresses in gameplay.

Limiting the Ship's Range

At this point the ship will disappear off either edge of the screen if you hold down an arrow key long enough. Let's correct this so the ship stops moving when it reaches the edge of the screen. We do this by modifying the `update()` method in `Ship`:

ship.py

```
    def update(self):
        """Update the ship's position based on movement flags."""
        # Update the ship's center value, not the rect.
❶       if self.moving_right and self.rect.right < self.screen_rect.right:
            self.center += self.ai_settings.ship_speed_factor
❷       if self.moving_left and self.rect.left > 0:
            self.center -= self.ai_settings.ship_speed_factor

        # Update rect object from self.center.
        self.rect.centerx = self.center
```

This code checks the position of the ship before changing the value of `self.center`. The code `self.rect.right` returns the x-coordinate value of the right edge of the ship's `rect`. If this value is less than the value returned by `self.screen_rect.right`, the ship hasn't reached the right edge of the screen ❶. The same goes for the left edge: if the value of the left side of the `rect` is greater than zero, the ship hasn't reached the left edge of the screen ❷. This ensures the ship is within these bounds before adjusting the value of `self.center`.

If you run *alien_invasion.py* now, the ship should stop moving at either edge of the screen.

Refactoring check_events()

The `check_events()` function will increase in length as we continue to develop the game, so let's break `check_events()` into two more functions: one that handles `KEYDOWN` events and another that handles `KEYUP` events:

game_functions.py

```
def check_keydown_events(event, ship):
    """Respond to keypresses."""
    if event.key == pygame.K_RIGHT:
        ship.moving_right = True
    elif event.key == pygame.K_LEFT:
        ship.moving_left = True

def check_keyup_events(event, ship):
    """Respond to key releases."""
    if event.key == pygame.K_RIGHT:
        ship.moving_right = False
    elif event.key == pygame.K_LEFT:
        ship.moving_left = False
```

```
def check_events(ship):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            check_keydown_events(event, ship)
        elif event.type == pygame.KEYUP:
            check_keyup_events(event, ship)
```

We make two new functions: `check_keydown_events()` and `check_keyup_events()`. Each needs an `event` parameter and a `ship` parameter. The bodies of these two functions are copied from `check_events()`, and we've replaced the old code with calls to the new functions. The `check_events()` function is simpler now with this cleaner code structure, which will make it easier to develop further responses to player input.

A Quick Recap

In the next section, we'll add the ability to shoot bullets, which involves a new file called *bullet.py* and some modifications to some of the files we already have. Right now, we have four files containing a number of classes, functions, and methods. To be clear about how the project is organized, let's review each of these files before adding more functionality.

alien_invasion.py

The main file, *alien_invasion.py*, creates a number of important objects used throughout the game: the settings are stored in `ai_settings`, the main display surface is stored in `screen`, and a `ship` instance is created in this file as well. Also stored in *alien_invasion.py* is the main loop of the game, which is a `while` loop that calls `check_events()`, `ship.update()`, and `update_screen()`.

alien_invasion.py is the only file you need to run when you want to play Alien Invasion. The other files—*settings.py*, *game_functions.py*, *ship.py*—contain code that is imported, directly or indirectly, into this file.

settings.py

The *settings.py* file contains the `Settings` class. This class only has an `__init__()` method, which initializes attributes controlling the game's appearance and the ship's speed.

game_functions.py

The *game_functions.py* file contains a number of functions that carry out the bulk of the work in the game. The `check_events()` function detects relevant events, such as keypresses and releases, and processes each of these types of events through the helper functions `check_keydown_events()` and

check_keyup_events(). For now, these functions manage the movement of the ship. The game_functions module also contains update_screen(), which redraws the screen on each pass through the main loop.

ship.py

The *ship.py* file contains the Ship class. Ship has an __init__() method, an update() method to manage the ship's position, and a blitme() method to draw the ship to the screen. The actual image of the ship is stored in *ship.bmp*, which is in the *images* folder.

TRY IT YOURSELF

12-3. Rocket: Make a game that begins with a rocket in the center of the screen. Allow the player to move the rocket up, down, left, or right using the four arrow keys. Make sure the rocket never moves beyond any edge of the screen.

12-4. Keys: Make a Pygame file that creates an empty screen. In the event loop, print the event.key attribute whenever a pygame.KEYDOWN event is detected. Run the program and press various keys to see how Pygame responds.

Shooting Bullets

Now let's add the ability to shoot bullets. We'll write code that fires a bullet (a small rectangle) when the player presses the spacebar. Bullets will then travel straight up the screen until they disappear off the top of the screen.

Adding the Bullet Settings

First, update *settings.py* to include the values we'll need for a new Bullet class, at the end of the __init__() method:

settings.py

```
    def __init__(self):
        --snip--
        # Bullet settings
        self.bullet_speed_factor = 1
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
```

These settings create dark gray bullets with a width of 3 pixels and a height of 15 pixels. The bullets will travel slightly slower than the ship.

Creating the Bullet Class

Now create a *bullet.py* file to store our `Bullet` class. Here's the first part of *bullet.py*:

bullet.py

```
import pygame
from pygame.sprite import Sprite

class Bullet(Sprite):
    """A class to manage bullets fired from the ship"""

    def __init__(self, ai_settings, screen, ship):
        """Create a bullet object at the ship's current position."""
        super(Bullet, self).__init__()
        self.screen = screen

        # Create a bullet rect at (0, 0) and then set correct position.
❶       self.rect = pygame.Rect(0, 0, ai_settings.bullet_width,
            ai_settings.bullet_height)
❷       self.rect.centerx = ship.rect.centerx
❸       self.rect.top = ship.rect.top

        # Store the bullet's position as a decimal value.
❹       self.y = float(self.rect.y)

❺       self.color = ai_settings.bullet_color
        self.speed_factor = ai_settings.bullet_speed_factor
```

The `Bullet` class inherits from `Sprite`, which we import from the `pygame.sprite` module. When you use sprites, you can group related elements in your game and act on all the grouped elements at once. To create a bullet instance, `__init__()` needs the `ai_settings`, `screen`, and `ship` instances, and we call `super()` to inherit properly from `Sprite`.

NOTE *The call `super(Bullet, self).__init__()` uses Python 2.7 syntax. This works in Python 3 too, or you can also write this call more simply as `super().__init__()`.*

At ❶, we create the bullet's `rect` attribute. The bullet is not based on an image so we have to build a rect from scratch using the `pygame.Rect()` class. This class requires the x- and y-coordinates of the top-left corner of the `rect`, and the width and height of the `rect`. We initialize the `rect` at (0, 0), but we'll move it to the correct location in the next two lines, because the bullet's position is dependent on the ship's position. We get the width and height of the bullet from the values stored in `ai_settings`.

At ❷, we set the bullet's `centerx` to be the same as the ship's `rect.centerx`. The bullet should emerge from the top of the ship, so we set the top of the bullet's `rect` to match the top of the ship's `rect`, making it look like the bullet is fired from the ship ❸.

We store a decimal value for the bullet's y-coordinate so we can make fine adjustments to the bullet's speed ❹. At ❺, we store the bullet's color and speed settings in `self.color` and `self.speed_factor`.

Here's the second part of *bullet.py*, `update()` and `draw_bullet()`:

bullet.py

```
    def update(self):
        """Move the bullet up the screen."""
        # Update the decimal position of the bullet.
❶       self.y -= self.speed_factor
        # Update the rect position.
❷       self.rect.y = self.y

    def draw_bullet(self):
        """Draw the bullet to the screen."""
❸       pygame.draw.rect(self.screen, self.color, self.rect)
```

The `update()` method manages the bullet's position. When a bullet is fired, it moves up the screen, which corresponds to a decreasing y-coordinate value; so to update the position, we subtract the amount stored in `self.speed_factor` from `self.y` ❶. We then use the value of `self.y` to set the value of `self.rect.y` ❷. The `speed_factor` attribute allows us to increase the speed of the bullets as the game progresses or as needed to refine the game's behavior. Once fired, a bullet's x-coordinate value never changes, so it will only travel vertically in a straight line.

When we want to draw a bullet, we'll call `draw_bullet()`. The `draw.rect()` function fills the part of the screen defined by the bullet's `rect` with the color stored in `self.color` ❸.

Storing Bullets in a Group

Now that we have a `Bullet` class and the necessary settings defined, we can write code to fire a bullet each time the player presses the spacebar. First, we'll create a group in *alien_invasion.py* to store all the live bullets so we can manage the bullets that have already been fired. This group will be an instance of the class `pygame.sprite.Group`, which behaves like a list with some extra functionality that's helpful when building games. We'll use this group to draw bullets to the screen on each pass through the main loop and to update each bullet's position:

alien_invasion.py

```
import pygame
from pygame.sprite import Group
--snip--

def run_game():
    --snip--
    # Make a ship.
    ship = Ship(ai_settings, screen)
    # Make a group to store bullets in.
❶   bullets = Group()

    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
```

```
❷          bullets.update()
           gf.update_screen(ai_settings, screen, ship, bullets)

run_game()
```

We import `Group` from `pygame.sprite`. At ❶, we make an instance of `Group` and call it `bullets`. This group is created outside of the `while` loop so we don't create a new group of bullets each time the loop cycles.

NOTE *If you make a group like this inside the loop, you'll be creating thousands of groups of bullets and your game will probably slow to a crawl. If your game freezes up, look carefully at what's happening in your main `while` loop.*

We pass `bullets` to `check_events()` and `update_screen()`. We'll need to work with `bullets` in `check_events()` when the spacebar is pressed, and we'll need to update the bullets that are being drawn to the screen in `update_screen()`.

When you call `update()` on a group ❷, the group automatically calls `update()` for each sprite in the group. The line `bullets.update()` calls `bullet.update()` for each bullet we place in the group `bullets`.

Firing Bullets

In *game_functions.py*, we need to modify `check_keydown_events()` to fire a bullet when the spacebar is pressed. We don't need to change `check_keyup_events()` because nothing happens when the key is released. We also need to modify `update_screen()` to make sure each bullet is redrawn to the screen before we call `flip()`. Here are the relevant changes to *game_functions.py*:

game_functions.py

```
--snip--
from bullet import Bullet

❶ def check_keydown_events(event, ai_settings, screen, ship, bullets):
      --snip--
❷     elif event.key == pygame.K_SPACE:
          # Create a new bullet and add it to the bullets group.
          new_bullet = Bullet(ai_settings, screen, ship)
          bullets.add(new_bullet)
      --snip--

❸ def check_events(ai_settings, screen, ship, bullets):
      """Respond to keypresses and mouse events."""
      for event in pygame.event.get():
          --snip--
          elif event.type == pygame.KEYDOWN:
              check_keydown_events(event, ai_settings, screen, ship, bullets)
          --snip--

❹ def update_screen(ai_settings, screen, ship, bullets):
      --snip--
      # Redraw all bullets behind ship and aliens.
❺     for bullet in bullets.sprites():
          bullet.draw_bullet()
```

```
    ship.blitme()
    --snip--
```

The group `bullets` is passed to `check_keydown_events()` ❶. When the player presses the spacebar, we create a new bullet (a `Bullet` instance that we name `new_bullet`) and add it to the group `bullets` ❷ using the `add()` method; the code `bullets.add(new_bullet)` stores the new bullet in the group `bullets`.

We need to add `bullets` as a parameter in the definition of `check_events()` ❸, and we need to pass `bullets` as an argument in the call to `check_keydown_events()` as well.

We give the `bullets` parameter to `update_screen()` at ❹, which draws the bullets to the screen. The `bullets.sprites()` method returns a list of all sprites in the group `bullets`. To draw all fired bullets to the screen, we loop through the sprites in `bullets` and call `draw_bullet()` on each one ❺.

If you run *alien_invasion.py* now, you should be able to move the ship right and left, and fire as many bullets as you want. The bullets travel up the screen and disappear when they reach the top, as shown in Figure 12-3. You can alter the size, color, and speed of the bullets in *settings.py*.

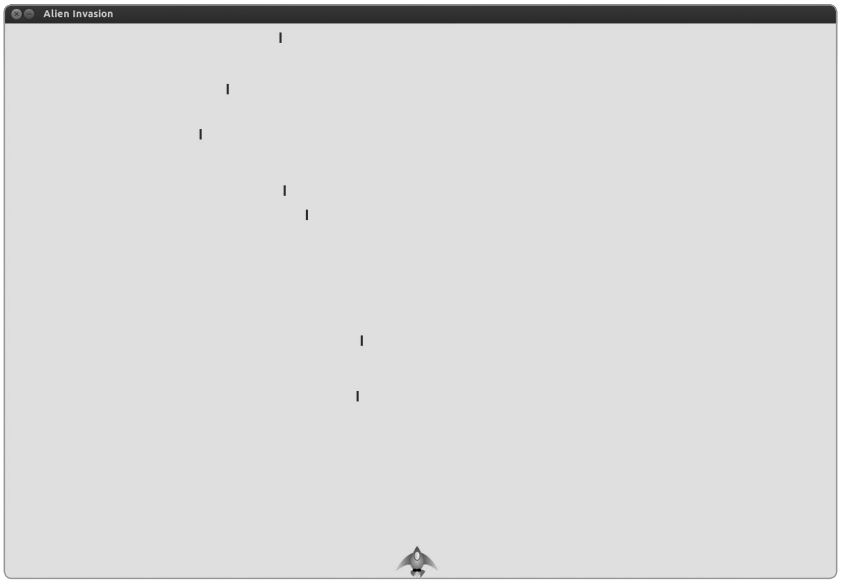


Figure 12-3: The ship after firing a series of bullets

Deleting Old Bullets

At the moment, the bullets disappear when they reach the top, but only because Pygame can't draw them above the top of the screen. The bullets actually continue to exist; their y-coordinate values just grow increasingly negative. This is a problem, because they continue to consume memory and processing power.

We need to get rid of these old bullets, or the game will slow down from doing so much unnecessary work. To do this, we need to detect when the `bottom` value of a bullet's `rect` has a value of 0, which indicates the bullet has passed off the top of the screen:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        bullets.update()

        # Get rid of bullets that have disappeared.
❶       for bullet in bullets.copy():
❷           if bullet.rect.bottom <= 0:
❸               bullets.remove(bullet)
❹       print(len(bullets))

        gf.update_screen(ai_settings, screen, ship, bullets)
```

You shouldn't remove items from a list or group within a `for` loop, so we have to loop over a copy of the group. We use the `copy()` method to set up the `for` loop ❶, which enables us to modify `bullets` inside the loop. We check each bullet to see whether it has disappeared off the top of the screen at ❷. If it has, we remove it from `bullets` ❸. At ❹ we insert a `print` statement to show how many bullets currently exist in the game and verify that they're being deleted.

If this code works correctly, we can watch the terminal output while firing bullets and see that the number of bullets decreases to zero after each set of bullets has cleared the top of the screen. After you run the game and verify that bullets are deleted properly, remove the `print` statement. If you leave it in, the game will slow down significantly because it takes more time to write output to the terminal than it does to draw graphics to the game window.

Limiting the Number of Bullets

Many shooting games limit the number of bullets a player can have on the screen at one time to encourage players to shoot accurately. We'll do the same in Alien Invasion.

First, store the number of bullets allowed in *settings.py*:

settings.py

```
        # Bullet settings
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
        self.bullets_allowed = 3
```

This limits the player to three bullets at a time. We'll use this setting in *game_functions.py* to check how many bullets exist before creating a new bullet in `check_keydown_events()`:

game_functions.py

```
def check_keydown_events(event, ai_settings, screen, ship, bullets):
    --snip--
    elif event.key == pygame.K_SPACE:
        # Create a new bullet and add it to the bullets group.
        if len(bullets) < ai_settings.bullets_allowed:
            new_bullet = Bullet(ai_settings, screen, ship)
            bullets.add(new_bullet)
```

When the spacebar is pressed, we check the length of `bullets`. If `len(bullets)` is less than three, we create a new bullet. But if three bullets are already active, nothing happens when the spacebar is pressed. If you run the game now, you should be able to fire bullets only in groups of three.

Creating the update_bullets() Function

We want to keep our main *alien_invasion.py* program file as simple as possible, so now that we've written and checked the bullet management code we can move it to the `game_functions` module. We'll create a new function called `update_bullets()` and add it to the end of *game_functions.py*:

game_functions.py

```
def update_bullets(bullets):
    """Update position of bullets and get rid of old bullets."""
    # Update bullet positions.
    bullets.update()

    # Get rid of bullets that have disappeared.
    for bullet in bullets.copy():
        if bullet.rect.bottom <= 0:
            bullets.remove(bullet)
```

The code for `update_bullets()` is cut and pasted from *alien_invasion.py*; the only parameter it needs is the group `bullets`.

The `while` loop in *alien_invasion.py* looks simple again:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
❶       gf.check_events(ai_settings, screen, ship, bullets)
❷       ship.update()
❸       gf.update_bullets(bullets)
❹       gf.update_screen(ai_settings, screen, ship, bullets)
```

We've made it so that our main loop contains only minimal code so we can quickly read the function names and understand what's happening in the game. The main loop checks for player input at ❶, and then it updates the position of the ship at ❷ and any bullets that have been fired at ❸. We then use the updated positions to draw a new screen at ❹.

Creating the fire_bullet() Function

Let's move the code for firing a bullet to a separate function so we can use a single line of code to fire a bullet and keep the `elif` block in `check_keydown_events()` simple:

game_functions.py

```
def check_keydown_events(event, ai_settings, screen, ship, bullets):
    """Respond to keypresses."""
    --snip--
    elif event.key == pygame.K_SPACE:
        fire_bullet(ai_settings, screen, ship, bullets)

def fire_bullet(ai_settings, screen, ship, bullets):
    """Fire a bullet if limit not reached yet."""
    # Create a new bullet and add it to the bullets group.
    if len(bullets) < ai_settings.bullets_allowed:
        new_bullet = Bullet(ai_settings, screen, ship)
        bullets.add(new_bullet)
```

The function `fire_bullet()` simply contains the code that was used to fire a bullet when the spacebar is pressed, and we add a call to `fire_bullet()` in `check_keydown_events()` when the spacebar is pressed.

Run *alien_invasion.py* one more time, and make sure you can still fire bullets without errors.

TRY IT YOURSELF

12-5. Sideways Shooter: Write a game that places a ship on the left side of the screen and allows the player to move the ship up and down. Make the ship fire a bullet that travels right across the screen when the player presses the spacebar. Make sure bullets are deleted once they disappear off the screen.

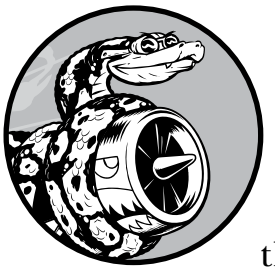
Summary

In this chapter, you learned to make a plan for a game. You learned the basic structure of a game written in Pygame. You learned to set a background color and store settings in a separate class where they can be made available to all parts of the game. You saw how to draw an image to the screen and give the player control over the movement of game elements. You learned to create elements that move on their own, like bullets flying up a screen, and how to delete objects that are no longer needed. You learned to refactor code in a project on a regular basis to facilitate ongoing development.

In Chapter 13, we'll add aliens to Alien Invasion. By the end of Chapter 13, you'll be able to shoot down aliens, hopefully before they reach your ship!

13

ALIENS!



In this chapter we'll add aliens to Alien Invasion. First, we'll add one alien near the top of the screen, and then we'll generate a whole fleet of aliens. We'll make the fleet advance sideways and down, and we'll get rid of any aliens hit by a bullet. Finally, we'll limit the number of ships a player has and end the game when the player runs out of ships.

As you work through this chapter, you'll learn more about Pygame and about managing a larger project. You'll also learn to detect collisions between game objects, like bullets and aliens. Detecting collisions helps you define interactions between elements in your games: you can confine a character inside the walls of a maze or pass a ball between two characters. We'll also continue to work from a plan that we revisit occasionally to maintain the focus of our code-writing sessions.

Before we start writing new code to add a fleet of aliens to the screen, let's look at the project and update our plan.

Reviewing Your Project

When you're beginning a new phase of development on a larger project, it's always a good idea to revisit your plan and clarify what you want to accomplish with the code you're about to write. In this chapter we will:

- Examine our code and determine if we need to refactor before implementing new features.
- Add a single alien to the top-left corner of the screen with appropriate spacing around it.
- Use the spacing around the first alien and the overall screen size to determine how many aliens can fit on the screen. We'll write a loop to create aliens to fill the upper portion of the screen.
- Make the fleet move sideways and down until the entire fleet is shot down, an alien hits the ship, or an alien reaches the ground. If the whole fleet is shot down, we'll create a new fleet. If an alien hits the ship or the ground, we'll destroy the ship and create a new fleet.
- Limit the number of ships the player can use, and end the game when the player has used up the allotment of ships.

We'll refine this plan as we implement features, but this is sufficient to start with.

You should also review code when you're about to begin working on a new series of features in a project. Because each new phase typically makes a project more complex, it's best to clean up cluttered or inefficient code.

Although we don't have much cleanup to do right now because we've been refactoring as we go, it's annoying to use the mouse to close the game each time we run it to test a new feature. Let's quickly add a keyboard shortcut to end the game when the user presses Q:

game_functions.py

```
def check_keydown_events(event, ai_settings, screen, ship, bullets):
    --snip--
    elif event.key == pygame.K_q:
        sys.exit()
```

In `check_keydown_events()` we add a new block that ends the game when Q is pressed. This is a fairly safe change because the Q key is far from the arrow keys and the spacebar, so it's unlikely a player will accidentally press Q and quit the game. Now, when testing, you can press Q to close the game rather than using your mouse to close the window.

Creating the First Alien

Placing one alien on the screen is like placing a ship on the screen. The behavior of each alien is controlled by a class called `Alien`, which we'll structure like the `Ship` class. We'll continue using bitmap images for simplicity. You can find your own image for an alien or use the one

shown in Figure 13-1, which is available in the book's resources through *https://www.nostarch.com/pythoncrashcourse/*. This image has a gray background, which matches the screen's background color. Make sure to save the image file you choose in the *images* folder.

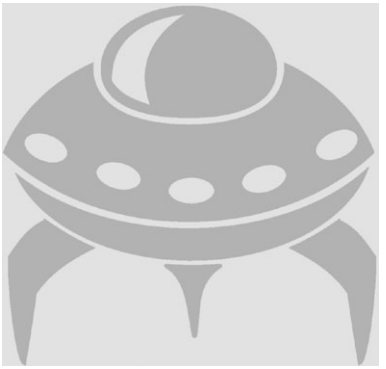


Figure 13-1: The alien we'll use to build the fleet

Creating the Alien Class

Now we'll write the `Alien` class:

alien.py

```
import pygame
from pygame.sprite import Sprite

class Alien(Sprite):
    """A class to represent a single alien in the fleet."""

    def __init__(self, ai_settings, screen):
        """Initialize the alien and set its starting position."""
        super(Alien, self).__init__()
        self.screen = screen
        self.ai_settings = ai_settings

        # Load the alien image and set its rect attribute.
        self.image = pygame.image.load('images/alien.bmp')
        self.rect = self.image.get_rect()

        # Start each new alien near the top left of the screen.
❶       self.rect.x = self.rect.width
        self.rect.y = self.rect.height

        # Store the alien's exact position.
        self.x = float(self.rect.x)

    def blitme(self):
        """Draw the alien at its current location."""
        self.screen.blit(self.image, self.rect)
```

Most of this class is like the `Ship` class except for the placement of the alien. We initially place each alien near the top-left corner of the screen, adding a space to the left of it that's equal to the alien's width and a space above it equal to its height ❶.

Creating an Instance of the Alien

Now we create an instance of `Alien` in *alien_invasion.py*:

alien_invasion.py

```
--snip--
from ship import Ship
from alien import Alien
import game_functions as gf

def run_game():
    --snip--
    # Make an alien.
    alien = Alien(ai_settings, screen)

    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(bullets)
        gf.update_screen(ai_settings, screen, ship, alien, bullets)

run_game()
```

Here we're importing the new `Alien` class and creating an instance of `Alien` just before entering the main `while` loop. Because we're not changing the alien's position yet, we aren't adding anything new inside the loop; however, we do modify the call to `update_screen()` to pass it the `alien` instance.

Making the Alien Appear Onscreen

To make the alien appear onscreen, we call its `blitme()` method in `update_screen()`:

game_functions.py

```
def update_screen(ai_settings, screen, ship, alien, bullets):
    --snip--

    # Redraw all bullets behind ship and aliens.
    for bullet in bullets:
        bullet.draw_bullet()
    ship.blitme()
    alien.blitme()

    # Make the most recently drawn screen visible.
    pygame.display.flip()
```

We draw the alien onscreen after the ship and the bullets have been drawn, so the aliens will be the top layer of the screen. Figure 13-2 shows the first alien on the screen.

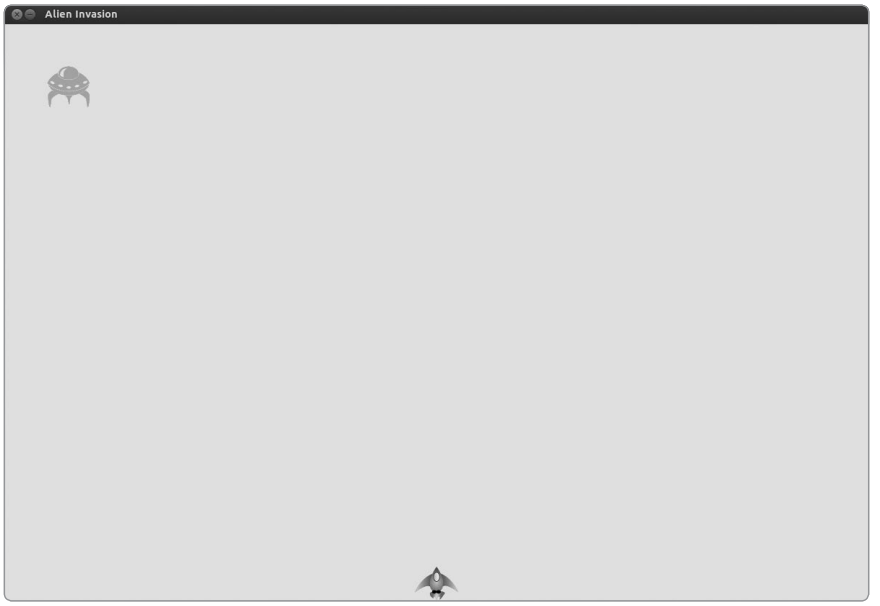


Figure 13-2: The first alien appears.

Now that the first alien appears correctly, we'll write the code to draw an entire fleet.

Building the Alien Fleet

To draw a fleet, we need to figure out how many aliens can fit across the screen and how many rows of aliens can fit down the screen. We'll first figure out the horizontal spacing between aliens and create a row; then we'll determine the vertical spacing and create an entire fleet.

Determining How Many Aliens Fit in a Row

To figure out how many aliens fit in a row, let's look at how much horizontal space we have. The screen width is stored in `ai_settings.screen_width`, but we need an empty margin on either side of the screen. We'll make this margin the width of one alien. Because we have two margins, the available space for aliens is the screen width minus two alien widths:

```
available_space_x = ai_settings.screen_width - (2 * alien_width)
```

We also need to set the spacing between aliens; we'll make it one alien width. The space needed to display one alien is twice its width: one width for the alien and one width for the empty space to its right. To find the number of aliens that fit across the screen, we divide the available space by two times the width of an alien:

```
number_aliens_x = available_space_x / (2 * alien_width)
```

We'll include these calculations when we create the fleet.

NOTE *One great aspect about calculations in programming is that you don't have to be sure your formula is correct when you first write it. You can try it out and see if it works. At worst, you'll have a screen that's overcrowded with aliens or has too few aliens. You can revise your calculation based on what you see on the screen.*

Creating Rows of Aliens

To create a row, first create an empty group called `aliens` in *alien_invasion.py* to hold all of our aliens, and then call a function in *game_functions.py* to create a fleet:

alien_invasion.py

```
import pygame
from pygame.sprite import Group
from settings import Settings
from ship import Ship
import game_functions as gf

def run_game():
    --snip--
    # Make a ship, a group of bullets, and a group of aliens.
    ship = Ship(ai_settings, screen)
    bullets = Group()
❶   aliens = Group()

    # Create the fleet of aliens.
❷   gf.create_fleet(ai_settings, screen, aliens)

    # Start the main loop for the game.
    while True:
        --snip--
❸       gf.update_screen(ai_settings, screen, ship, aliens, bullets)

run_game()
```

Because we're no longer creating aliens directly in *alien_invasion.py*, we don't need to import the `Alien` class into this file.

Create an empty group to hold all of the aliens in the game ❶. Then, call the new function `create_fleet()` ❷, which we'll write shortly, and pass it the `ai_settings`, the `screen` object, and the empty group `aliens`. Next, modify the call to `update_screen()` to give it access to the group of aliens ❸.

We also need to modify `update_screen()`:

game_functions.py

```
def update_screen(ai_settings, screen, ship, aliens, bullets):
    --snip--
    ship.blitme()
    aliens.draw(screen)

    # Make the most recently drawn screen visible.
    pygame.display.flip()
```

When you call `draw()` on a group, Pygame automatically draws each element in the group at the position defined by its `rect` attribute. In this case, `aliens.draw(screen)` draws each alien in the group to the screen.

Creating the Fleet

Now we can create the fleet. Here's the new function `create_fleet()`, which we place at the end of *game_functions.py*. We also need to import the `Alien` class, so make sure you add an `import` statement at the top of the file:

game_functions.py

```
--snip--
from bullet import Bullet
from alien import Alien
--snip--

def create_fleet(ai_settings, screen, aliens):
    """Create a full fleet of aliens."""
    # Create an alien and find the number of aliens in a row.
    # Spacing between each alien is equal to one alien width.
❶   alien = Alien(ai_settings, screen)
❷   alien_width = alien.rect.width
❸   available_space_x = ai_settings.screen_width - 2 * alien_width
❹   number_aliens_x = int(available_space_x / (2 * alien_width))

    # Create the first row of aliens.
❺   for alien_number in range(number_aliens_x):
        # Create an alien and place it in the row.
❻       alien = Alien(ai_settings, screen)
        alien.x = alien_width + 2 * alien_width * alien_number
        alien.rect.x = alien.x
        aliens.add(alien)
```

We've already thought through most of this code. We need to know the alien's width and height in order to place aliens, so we create an alien at ❶ before we perform calculations. This alien won't be part of the fleet, so don't add it to the group `aliens`. At ❷ we get the alien's width from its `rect` attribute and store this value in `alien_width` so we don't have to keep working through the `rect` attribute. At ❸ we calculate the horizontal space available for aliens and the number of aliens that can fit into that space.

The only change here from our original formulas is that we're using `int()` to ensure we end up with an integer number of aliens ❹ because we don't want to create partial aliens, and the `range()` function needs an

integer. The `int()` function drops the decimal part of a number, effectively rounding down. (This is helpful because we'd rather have a little extra space in each row than an overly crowded row.)

Next, set up a loop that counts from 0 to the number of aliens we need to make ❺. In the main body of the loop, create a new alien and then set its x-coordinate value to place it in the row ❻. Each alien is pushed to the right one alien width from the left margin. Next, we multiply the alien width by 2 to account for the space each alien takes up, including the empty space to its right, and we multiply this amount by the alien's position in the row. Then we add each new alien to the group `aliens`.

When you run Alien Invasion, you should see the first row of aliens appear, as in Figure 13-3.

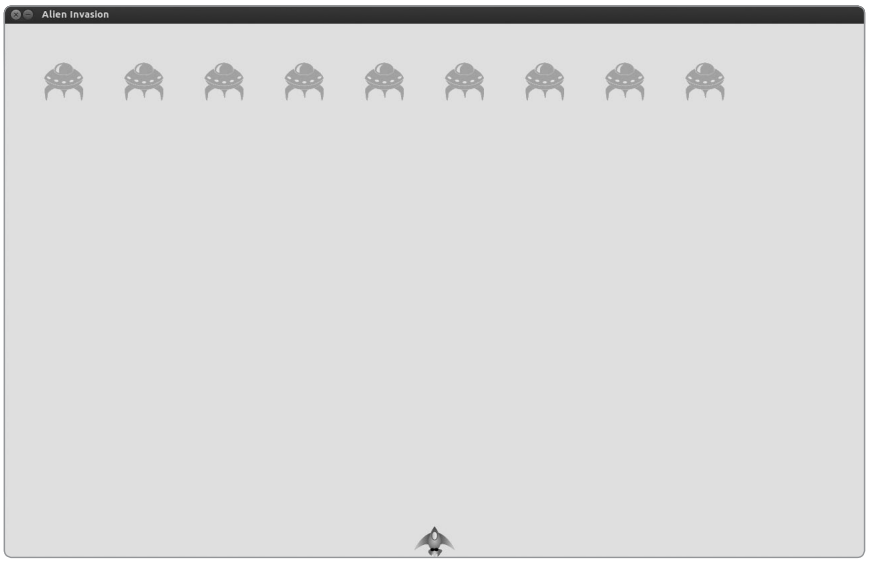


Figure 13-3: The first row of aliens

The first row is offset to the left, which is actually good for gameplay because we want the fleet to move right until it hits the edge of the screen, then drop down a bit, then move left, and so forth. Like the classic game *Space Invaders*, this movement is more interesting than having the fleet drop straight down. We'll continue this motion until all aliens are shot down or until an alien hits the ship or the bottom of the screen.

NOTE *Depending on the screen width you've chosen, the alignment of the first row of aliens may look slightly different on your system.*

Refactoring create_fleet()

If we were finished creating a fleet, we'd probably leave `create_fleet()` as is, but we have more work to do, so let's clean up the function a bit. Here's `create_fleet()` with two new functions: `get_number_aliens_x()` and `create_alien()`:

game_functions.py

```
❶ def get_number_aliens_x(ai_settings, alien_width):
    """Determine the number of aliens that fit in a row."""
    available_space_x = ai_settings.screen_width - 2 * alien_width
    number_aliens_x = int(available_space_x / (2 * alien_width))
    return number_aliens_x

def create_alien(ai_settings, screen, aliens, alien_number):
    """Create an alien and place it in the row."""
    alien = Alien(ai_settings, screen)
❷   alien_width = alien.rect.width
    alien.x = alien_width + 2 * alien_width * alien_number
    alien.rect.x = alien.x
    aliens.add(alien)

def create_fleet(ai_settings, screen, aliens):
    """Create a full fleet of aliens."""
    # Create an alien and find the number of aliens in a row.
    alien = Alien(ai_settings, screen)
❸   number_aliens_x = get_number_aliens_x(ai_settings, alien.rect.width)

    # Create the first row of aliens.
    for alien_number in range(number_aliens_x):
❹       create_alien(ai_settings, screen, aliens, alien_number)
```

The body of `get_number_aliens_x()` is exactly as it was in `create_fleet()` ❶. The body of `create_alien()` is also unchanged from `create_fleet()` except that we use the alien that was just created to get the alien width ❷. At ❸ we replace the code for determining the horizontal spacing with a call to `get_number_aliens_x()`, and we remove the line referring to `alien_width`, because that's now handled inside `create_alien()`. At ❹ we call `create_alien()`. This refactoring will make it easier to add new rows and create an entire fleet.

Adding Rows

To finish the fleet, determine the number of rows that fit on the screen and then repeat the loop (for creating the aliens in one row) that number of times. To determine the number of rows, we find the available vertical space by subtracting the alien height from the top, the ship height from the bottom, and two alien heights from the bottom of the screen:

```
available_space_y = ai_settings.screen_height - 3 * alien_height - ship_height
```

The result will create some empty space above the ship, so the player has some time to start shooting aliens at the beginning of each level.

Each row needs some empty space below it, which we'll make equal to the height of one alien. To find the number of rows, we divide the available space by two times the height of an alien. (Again, if these calculations are off, we'll see it right away and adjust until we have reasonable spacing.)

```
number_rows = available_height_y / (2 * alien_height)
```

Now that we know how many rows fit in a fleet, we can repeat the code for creating a row:

game_functions.py

```
❶ def get_number_rows(ai_settings, ship_height, alien_height):
    """Determine the number of rows of aliens that fit on the screen."""
❷   available_space_y = (ai_settings.screen_height -
                            (3 * alien_height) - ship_height)
    number_rows = int(available_space_y / (2 * alien_height))
    return number_rows

def create_alien(ai_settings, screen, aliens, alien_number, row_number):
    --snip--
    alien.x = alien_width + 2 * alien_width * alien_number
    alien.rect.x = alien.x
❸   alien.rect.y = alien.rect.height + 2 * alien.rect.height * row_number
    aliens.add(alien)

def create_fleet(ai_settings, screen, ship, aliens):
    --snip--
    number_aliens_x = get_number_aliens_x(ai_settings, alien.rect.width)
    number_rows = get_number_rows(ai_settings, ship.rect.height,
        alien.rect.height)

    # Create the fleet of aliens.
❹   for row_number in range(number_rows):
        for alien_number in range(number_aliens_x):
            create_alien(ai_settings, screen, aliens, alien_number,
                row_number)
```

To calculate the number of rows we can fit on the screen, we write our `available_space_y` and `number_rows` calculations into the function `get_number_rows()` ❶, which is similar to `get_number_aliens_x()`. The calculation is wrapped in parentheses so the outcome can be split over two lines, which results in lines of 79 characters or less as is recommended ❷. We use `int()` because we don't want to create a partial row of aliens.

To create multiple rows, we use two nested loops: one outer and one inner loop ❹. The inner loop creates the aliens in one row. The outer loop counts from 0 to the number of rows we want; Python will use the code for making a single row and repeat it `number_rows` times.

To nest the loops, write the new `for` loop and indent the code you want to repeat. (Most text editors make it easy to indent and unindent blocks of code, but for help see Appendix B.) Now when we call `create_alien()`, we include an argument for the row number so each row can be placed farther down the screen.

The definition of `create_alien()` needs a parameter to hold the row number. Within `create_alien()`, we change an alien's y-coordinate value when it's not in the first row ❸ by starting with one alien's height to create empty space at the top of the screen. Each row starts two alien heights below the last row, so we multiply the alien height by two and then by the row number. The first row number is 0, so the vertical placement of the first row is unchanged. All subsequent rows are placed farther down the screen.

The definition of `create_fleet()` also has a new parameter for the `ship` object, which means we need to include the `ship` argument in the call to `create_fleet()` in *alien_invasion.py*:

alien_invasion.py

```
# Create the fleet of aliens.
gf.create_fleet(ai_settings, screen, ship, aliens)
```

When you run the game now, you should see a fleet of aliens, as in Figure 13-4.

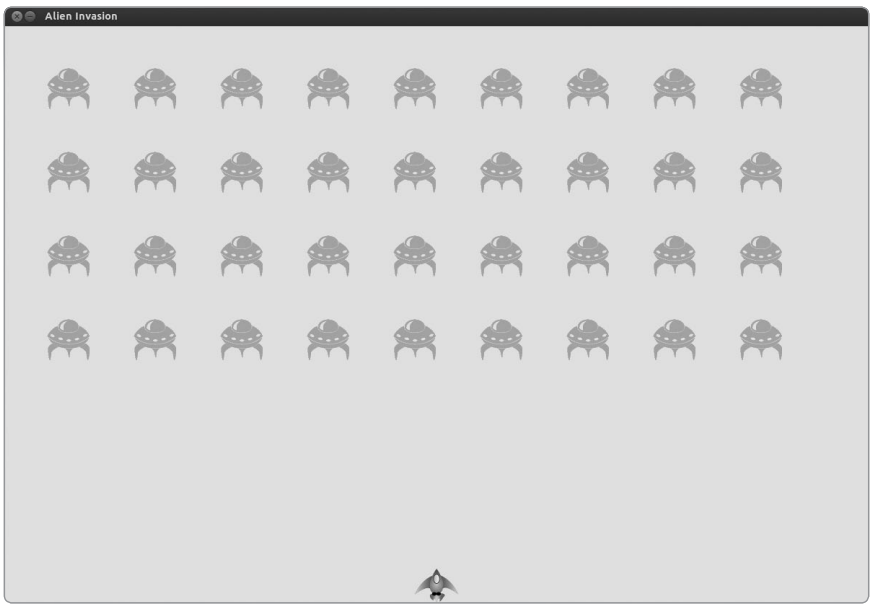


Figure 13-4: The full fleet appears.

In the next section, we'll make the fleet move!

TRY IT YOURSELF

13-1. Stars: Find an image of a star. Make a grid of stars appear on the screen.

13-2. Better Stars: You can make a more realistic star pattern by introducing randomness when you place each star. Recall that you can get a random number like this:

```
from random import randint
random_number = randint(-10,10)
```

This code returns a random integer between –10 and 10. Using your code in Exercise 13-1, adjust each star's position by a random amount.

Making the Fleet Move

Now let's make our fleet of aliens move to the right across the screen until it hits the edge, and then make it drop a set amount and move in the other direction. We'll continue this movement until all aliens have been shot down, one collides with the ship, or one reaches the bottom of the screen. Let's begin by making the fleet move to the right.

Moving the Aliens Right

To move the aliens, we'll use an `update()` method in *alien.py*, which we'll call for each alien in the group of aliens. First, add a setting to control the speed of each alien:

settings.py

```
    def __init__(self):
        --snip--
        # Alien settings
        self.alien_speed_factor = 1
```

Then, use this setting to implement `update()`:

alien.py

```
    def update(self):
        """Move the alien right."""
❶       self.x += self.ai_settings.alien_speed_factor
❷       self.rect.x = self.x
```

Each time we update an alien's position, we move it to the right by the amount stored in `alien_speed_factor`. We track the alien's exact position with the `self.x` attribute, which can hold decimal values ❶. We then use the value of `self.x` to update the position of the alien's `rect` ❷.

In the main `while` loop, we have calls to update the ship and bullets. Now we need to update the position of each alien as well:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(bullets)
        gf.update_aliens(aliens)
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

We update the aliens' positions after the bullets have been updated, because we'll soon be checking to see whether any bullets hit any aliens.

Finally, add the new function `update_aliens()` at the end of the file *game_functions.py*:

game_functions.py

```
def update_aliens(aliens):
    """Update the postions of all aliens in the fleet."""
    aliens.update()
```

We use the `update()` method on the `aliens` group, which automatically calls each alien's `update()` method. When you run Alien Invasion now, you should see the fleet move right and disappear off the side of the screen.

Creating Settings for Fleet Direction

Now we'll create the settings that will make the fleet move down the screen and to the left when it hits the right edge of the screen. Here's how to implement this behavior:

settings.py

```
        # Alien settings
        self.alien_speed_factor = 1
        self.fleet_drop_speed = 10
        # fleet_direction of 1 represents right; -1 represents left.
        self.fleet_direction = 1
```

The setting `fleet_drop_speed` controls how quickly the fleet drops down the screen each time an alien reaches either edge. It's helpful to separate this speed from the aliens' horizontal speed so you can adjust the two speeds independently.

To implement the setting `fleet_direction`, we could use a text value, such as `'left'` or `'right'`, but we'd end up with `if-elif` statements testing for the fleet direction. Instead, because we have only two directions to deal with, let's use the values 1 and –1 and switch between them each time the fleet changes direction. (Using numbers also makes sense because moving right involves adding to each alien's x-coordinate value, and moving left involves subtracting from each alien's x-coordinate value.)

Checking to See Whether an Alien Has Hit the Edge

Now we need a method to check whether an alien is at either edge, and we need to modify `update()` to allow each alien to move in the appropriate direction:

alien.py

```
    def check_edges(self):
        """Return True if alien is at edge of screen."""
        screen_rect = self.screen.get_rect()
❶       if self.rect.right >= screen_rect.right:
            return True
❷       elif self.rect.left <= 0:
            return True

    def update(self):
        """Move the alien right or left."""
❸       self.x += (self.ai_settings.alien_speed_factor *
                   self.ai_settings.fleet_direction)
        self.rect.x = self.x
```

We can call the new method `check_edges()` on any alien to see if it's at the left or right edge. The alien is at the right edge if the `right` attribute of its `rect` is greater than or equal to the `right` attribute of the screen's `rect` ❶. It's at the left edge if its `left` value is less than or equal to 0 ❷.

We modify the method `update()` to allow motion to the left or right ❸ by multiplying the alien's speed factor by the value of `fleet_direction`. If `fleet_direction` is 1, the value of `alien_speed_factor` will be added to the alien's current position, moving the alien to the right; if `fleet_direction` is –1, the value will be subtracted from the alien's position, moving the alien to the left.

Dropping the Fleet and Changing Direction

When an alien reaches the edge, the entire fleet needs to drop down and change direction. We therefore need to make some substantial changes in *game_functions.py* because that's where we check to see if any aliens are at the left or right edge. We'll make this happen by writing the functions `check_fleet_edges()` and `change_fleet_direction()`, and then modifying `update_aliens()`:

game_functions.py

```
def check_fleet_edges(ai_settings, aliens):
    """Respond appropriately if any aliens have reached an edge."""
❶   for alien in aliens.sprites():
        if alien.check_edges():
            change_fleet_direction(ai_settings, aliens)
            break
```

```
def change_fleet_direction(ai_settings, aliens):
    """Drop the entire fleet and change the fleet's direction."""
    for alien in aliens.sprites():
❷       alien.rect.y += ai_settings.fleet_drop_speed
    ai_settings.fleet_direction *= -1

def update_aliens(ai_settings, aliens):
    """
    Check if the fleet is at an edge,
      and then update the postions of all aliens in the fleet.
    """
❸   check_fleet_edges(ai_settings, aliens)
    aliens.update()
```

In `check_fleet_edges()`, we loop through the fleet and call `check_edges()` on each alien ❶. If `check_edges()` returns `True`, we know an alien is at an edge and the whole fleet needs to change direction, so we call `change_fleet_direction()` and break out of the loop. In `change_fleet_direction()`, we loop through all the aliens and drop each one using the setting `fleet_drop_speed` ❷; then we change the value of `fleet_direction` by multiplying its current value by –1.

We've modified the function `update_aliens()` to determine whether any aliens are at an edge by calling `check_fleet_edges()` ❸. This function needs an `ai_settings` parameter, so we include an argument for `ai_settings` in the call to `update_aliens()`:

alien_invasion.py

```
# Start the main loop for the game.
while True:
    gf.check_events(ai_settings, screen, ship, bullets)
    ship.update()
    gf.update_bullets(bullets)
    gf.update_aliens(ai_settings, aliens)
    gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

If you run the game now, the fleet should move back and forth between the edges of the screen and drop down every time it hits an edge. Now we can begin shooting down aliens and watch for any aliens that hit the ship or reach the bottom of the screen.

TRY IT YOURSELF

13-3. Raindrops: Find an image of a raindrop and create a grid of raindrops. Make the raindrops fall toward the bottom of the screen until they disappear.

13-4. Steady Rain: Modify your code in Exercise 13-3 so that when a row of raindrops disappears off the bottom of the screen, a new row appears at the top of the screen and begins to fall.

Shooting Aliens

We've built our ship and a fleet of aliens, but when the bullets reach the aliens, they simply pass through because we aren't checking for collisions. In game programming, *collisions* happen when game elements overlap. To make the bullets shoot down aliens, we'll use the method `sprite.groupcollide()` to look for collisions between members of two groups.

Detecting Bullet Collisions

We want to know right away when a bullet hits an alien so we can make an alien disappear as soon as it's hit. To do this, we'll look for collisions immediately after updating a bullet's position.

The `sprite.groupcollide()` method compares each bullet's `rect` with each alien's `rect` and returns a dictionary containing the bullets and aliens that have collided. Each key in the dictionary is a bullet, and the corresponding value is the alien that was hit. (We'll use this dictionary when we implement a scoring system in Chapter 14.)

Use this code to check for collisions in the `update_bullets()` function:

game_functions.py

```
def update_bullets(aliens, bullets):
    """Update position of bullets and get rid of old bullets."""
    --snip--
    # Check for any bullets that have hit aliens.
    # If so, get rid of the bullet and the alien.
    collisions = pygame.sprite.groupcollide(bullets, aliens, True, True)
```

The new line we added loops through each bullet in the group `bullets` and then loops through each alien in the group `aliens`. Whenever the `rects` of a bullet and alien overlap, `groupcollide()` adds a key-value pair to the dictionary it returns. The two `True` arguments tell Pygame whether to delete the bullets and aliens that have collided. (To make a high-powered bullet that's able to travel to the top of the screen, destroying every alien in its path, you could set the first Boolean argument to `False` and keep the second Boolean argument set to `True`. The aliens hit would disappear, but all bullets would stay active until they disappeared off the top of the screen.)

We pass the argument `aliens` in the call to `update_bullets()`:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(aliens, bullets)
        gf.update_aliens(ai_settings, aliens)
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

When you run Alien Invasion now, aliens you hit should disappear. Figure 13-5 shows a fleet that has been partially shot down.

Figure 13-5: We can shoot aliens!

Making Larger Bullets for Testing

You can test many features of the game simply by running the game, but some features are tedious to test in the normal version of a game. For example, it's a lot of work to shoot down every alien on the screen multiple times to test if your code responds to an empty fleet correctly.

To test particular features, you can change certain game settings to focus on a particular area. For example, you might shrink the screen so there are fewer aliens to shoot down or increase the bullet speed and give yourself lots of bullets at once.

My favorite change for testing Alien Invasion is to use superwide bullets that remain active even after they've hit an alien (see Figure 13-6). Try setting `bullet_width` to 300 to see how quickly you can shoot down the fleet!

Changes like these will help you test the game more efficiently and possibly spark ideas for giving players bonus powers. (Just remember to restore the settings to normal once you're finished testing a feature.)

Figure 13-6: Extra-powerful bullets make some aspects of the game easier to test.

Repopulating the Fleet

One key feature of Alien Invasion is that the aliens are relentless: every time the fleet is destroyed, a new fleet should appear.

To make a new fleet of aliens appear after a fleet has been destroyed, first check to see whether the group `aliens` is empty. If it is, we call `create_fleet()`. We'll perform this check in `update_bullets()` because that's where individual aliens are destroyed:

game_functions.py

```
def update_bullets(ai_settings, screen, ship, aliens, bullets):
    --snip--
    # Check for any bullets that have hit aliens.
    # If so, get rid of the bullet and the alien.
    collisions = pygame.sprite.groupcollide(bullets, aliens, True, True)

❶   if len(aliens) == 0:
        # Destroy existing bullets and create new fleet.
❷       bullets.empty()
        create_fleet(ai_settings, screen, ship, aliens)
```

At ❶ we check whether the group `aliens` is empty. If it is, we get rid of any existing bullets by using the `empty()` method, which removes all the remaining sprites from a group ❷. We also call `create_fleet()`, which fills the screen with aliens again.

The definition of `update_bullets()` now has the additional parameters `ai_settings`, `screen`, and `ship`, so we need to update the call to `update_bullets()` in *alien_invasion.py*:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
        gf.update_aliens(ai_settings, aliens)
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Now a new fleet appears as soon as you destroy the current fleet.

Speeding Up the Bullets

If you've tried firing at the aliens in the game's current state, you may have noticed that the bullets have slowed down a bit. This is because Pygame is now doing more work on each pass through the loop. We can increase the speed of the bullets by adjusting the value of `bullet_speed_factor` in *settings.py*. If we increase this value (to 3, for example), the bullets should travel up the screen at a reasonable speed again:

settings.py

```
        # Bullet settings
        self.bullet_speed_factor = 3
        self.bullet_width = 3
        --snip--
```

The best value for this setting depends on the speed of your system, so find a value that works for you.

Refactoring update_bullets()

Let's refactor `update_bullets()` so it's not doing so many different tasks. We'll move the code for dealing with bullet-alien collisions to a separate function:

game_functions.py

```
def update_bullets(ai_settings, screen, ship, aliens, bullets):
    --snip--
    # Get rid of bullets that have disappeared.
    for bullet in bullets.copy():
        if bullet.rect.bottom <= 0:
            bullets.remove(bullet)

    check_bullet_alien_collisions(ai_settings, screen, ship, aliens, bullets)

def check_bullet_alien_collisions(ai_settings, screen, ship, aliens, bullets):
    """Respond to bullet-alien collisions."""
    # Remove any bullets and aliens that have collided.
    collisions = pygame.sprite.groupcollide(bullets, aliens, True, True)
```

```
    if len(aliens) == 0:
        # Destroy existing bullets and create new fleet.
        bullets.empty()
        create_fleet(ai_settings, screen, ship, aliens)
```

We've created a new function, `check_bullet_alien_collisions()`, to look for collisions between bullets and aliens, and to respond appropriately if the entire fleet has been destroyed. This keeps `update_bullets()` from growing too long and simplifies further development.

TRY IT YOURSELF

13-5. Catch: Create a game that places a character that you can move left and right at the bottom of the screen. Make a ball appear at a random position at the top of the screen and fall down the screen at a steady rate. If your character "catches" the ball by colliding with it, make the ball disappear. Make a new ball each time your character catches the ball or whenever the ball disappears off the bottom of the screen.

Ending the Game

What's the fun and challenge in a game if you can't lose? If the player doesn't shoot down the fleet quickly enough, we'll have the aliens destroy the ship if they hit it. At the same time, we'll limit the number of ships a player can use and we'll destroy the ship when an alien reaches the bottom of the screen. We'll end the game when the player has used up all their ships.

Detecting Alien-Ship Collisions

We'll start by checking for collisions between aliens and the ship so we can respond appropriately when an alien hits it. We'll check for alien-ship collisions immediately after updating the position of each alien:

game_functions.py

```
def update_aliens(ai_settings, ship, aliens):
    """
    Check if the fleet is at an edge,
      and then update the postions of all aliens in the fleet.
    """
    check_fleet_edges(ai_settings, aliens)
    aliens.update()

    # Look for alien-ship collisions.
❶   if pygame.sprite.spritecollideany(ship, aliens):
❷       print("Ship hit!!!")
```

The method `spritecollideany()` takes two arguments: a sprite and a group. The method looks for any member of the group that's collided with the sprite and stops looping through the group as soon as it finds one member that has collided with the sprite. Here, it loops through the group `aliens` and returns the first alien it finds that has collided with `ship`.

If no collisions occur, `spritecollideany()` returns `None` and the `if` block at ❶ won't execute. If it finds an alien that's collided with the ship, it returns that alien and the `if` block executes: it prints *Ship hit!!!* ❷. (When an alien hits the ship, we'll need to do a number of tasks: we'll need to delete all remaining aliens and bullets, recenter the ship, and create a new fleet. Before we write code to do all this, we need to know that our approach for detecting alien-ship collisions works correctly. Writing a `print` statement is a simple way to ensure we're detecting collisions properly.)

Now we need to pass `ship` to `update_aliens()`:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
        gf.update_aliens(ai_settings, ship, aliens)
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Now when you run Alien Invasion, *Ship hit!!!* should appear in the terminal whenever an alien runs into the ship. When testing this feature, set `alien_drop_speed` to a higher value such as 50 or 100 so that the aliens will reach your ship faster.

Responding to Alien-Ship Collisions

Now we need to figure out what happens when an alien collides with the ship. Instead of destroying the `ship` instance and creating a new one, we'll count how many times the ship has been hit by tracking statistics for the game. (Tracking statistics will also be useful for scoring.)

Let's write a new class, `GameStats`, to track game statistics, and save it as *game_stats.py*:

game_stats.py

```
class GameStats():
    """Track statistics for Alien Invasion."""

    def __init__(self, ai_settings):
        """Initialize statistics."""
        self.ai_settings = ai_settings
❶       self.reset_stats()

    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.ai_settings.ship_limit
```

We'll make one `GameStats` instance for the entire time Alien Invasion is running, but we'll need to reset some statistics each time the player starts

a new game. To do this, we'll initialize most of the statistics in the method `reset_stats()` instead of directly in `__init__()`. We'll call this method from `__init__()` so the statistics are set properly when the `GameStats` instance is first created ❶, but we'll also be able to call `reset_stats()` any time the player starts a new game.

Right now we have only one statistic, `ships_left`, the value of which will change throughout the game. The number of ships the player starts with is stored in *settings.py* as `ship_limit`:

settings.py

```
        # Ship settings
        self.ship_speed_factor = 1.5
        self.ship_limit = 3
```

We also need to make a few changes in *alien_invasion.py*, to create an instance of `GameStats`:

alien_invasion.py

```
--snip--
from settings import Settings
❶ from game_stats import GameStats
--snip--

def run_game():
    --snip--
    pygame.display.set_caption("Alien Invasion")

    # Create an instance to store game statistics.
❷   stats = GameStats(ai_settings)
    --snip--
    # Start the main loop for the game.
    while True:
        --snip--
        gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
❸       gf.update_aliens(ai_settings, stats, screen, ship, aliens, bullets)
        --snip--
```

We import the new `GameStats` class ❶, make a `stats` instance ❷, and then add the `stats`, `screen`, and `ship` arguments in the call to `update_aliens()` ❸. We'll use these arguments to track the number of ships the player has left and to build a new fleet when an alien hits the ship.

When an alien hits the ship, we subtract one from the number of ships left, destroy all existing aliens and bullets, create a new fleet, and reposition the ship in the middle of the screen. (We'll also pause the game for a moment so the player can notice the collision and regroup before a new fleet appears.)

Let's put most of this code in the function `ship_hit()`:

game_functions.py

```
import sys
❶ from time import sleep

import pygame
--snip--
```

```
def ship_hit(ai_settings, stats, screen, ship, aliens, bullets):
    """Respond to ship being hit by alien."""
    # Decrement ships_left.
❷   stats.ships_left -= 1

    # Empty the list of aliens and bullets.
❸   aliens.empty()
    bullets.empty()

    # Create a new fleet and center the ship.
❹   create_fleet(ai_settings, screen, ship, aliens)
    ship.center_ship()

    # Pause.
❺   sleep(0.5)

❻ def update_aliens(ai_settings, stats, screen, ship, aliens, bullets):
    --snip--
    # Look for alien-ship collisions.
    if pygame.sprite.spritecollideany(ship, aliens):
        ship_hit(ai_settings, stats, screen, ship, aliens, bullets)
```

We first import the `sleep()` function from the `time` module to pause the game ❶. The new function `ship_hit()` coordinates the response when the ship is hit by an alien. Inside `ship_hit()`, the number of ships left is reduced by 1 ❷, after which we empty the groups `aliens` and `bullets` ❸.

Next, we create a new fleet and center the ship ❹. (We'll add the method `center_ship()` to `Ship` in a moment.) Finally, we pause after the updates have been made to all the game elements but before any changes have been drawn to the screen so the player can see that their ship has been hit ❺. The screen will freeze momentarily, and the player will see that the alien has hit the ship. When the `sleep()` function ends, the code will move on to the `update_screen()` function, which will draw the new fleet to the screen.

We also update the definition of `update_aliens()` to include the parameters `stats`, `screen`, and `bullets` ❻ so it can pass these values in the call to `ship_hit()`.

Here's the new method `center_ship()`; add it to the end of *ship.py*:

ship.py

```
    def center_ship(self):
        """Center the ship on the screen."""
        self.center = self.screen_rect.centerx
```

To center the ship, we set the value of the ship's `center` attribute to match the center of the screen, which we get through the `screen_rect` attribute.

NOTE *Notice that we never make more than one ship; we make only one ship instance for the whole game and recenter it whenever the ship has been hit. The statistic* `ships_left` *will tell us when the player has run out of ships.*

Run the game, shoot a few aliens, and let an alien hit the ship. The game should pause, and a new fleet should appear with the ship centered at the bottom of the screen again.

Aliens that Reach the Bottom of the Screen

If an alien reaches the bottom of the screen, we'll respond the same way we do when an alien hits the ship. Add a new function to perform this check, and call it from `update_aliens()`:

game_functions.py

```
def check_aliens_bottom(ai_settings, stats, screen, ship, aliens, bullets):
    """Check if any aliens have reached the bottom of the screen."""
    screen_rect = screen.get_rect()
    for alien in aliens.sprites():
❶       if alien.rect.bottom >= screen_rect.bottom:
            # Treat this the same as if the ship got hit.
            ship_hit(ai_settings, stats, screen, ship, aliens, bullets)
            break

def update_aliens(ai_settings, stats, screen, ship, aliens, bullets):
    --snip--
    # Look for aliens hitting the bottom of the screen.
❷   check_aliens_bottom(ai_settings, stats, screen, ship, aliens, bullets)
```

The function `check_aliens_bottom()` checks to see whether any aliens have reached the bottom of the screen. An alien reaches the bottom when its `rect.bottom` value is greater than or equal to the screen's `rect.bottom` attribute ❶. If an alien reaches the bottom, we call `ship_hit()`. If one alien hits the bottom, there's no need to check the rest, so we break out of the loop after calling `ship_hit()`.

We call `check_aliens_bottom()` after updating the positions of all the aliens and after looking for alien-ship collisions ❷. Now a new fleet will appear every time the ship is hit by an alien or an alien reaches the bottom of the screen.

Game Over!

Alien Invasion feels more complete now, but the game never ends. The value of `ships_left` just grows increasingly negative. Let's add a `game_active` flag as an attribute to `GameStats` to end the game when the player runs out of ships:

game_stats.py

```
    def __init__(self, settings):
        --snip--
        # Start Alien Invasion in an active state.
        self.game_active = True
```

Now we add code to `ship_hit()` that sets `game_active` to `False` if the player has used up all their ships:

game_functions.py

```
def ship_hit(ai_settings, stats, screen, ship, aliens, bullets):
    """Respond to ship being hit by alien."""
    if stats.ships_left > 0:
        # Decrement ships_left.
        stats.ships_left -= 1
        --snip--
        # Pause.
        sleep(0.5)

    else:
        stats.game_active = False
```

Most of `ship_hit()` is unchanged. We've moved all of the existing code into an `if` block, which tests to make sure the player has at least one ship remaining. If so, we create a new fleet, pause, and move on. If the player has no ships left, we set `game_active` to `False`.

Identifying When Parts of the Game Should Run

In *alien_invasion.py* we need to identify the parts of the game that should always run and the parts that should run only when the game is active:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, ship, bullets)

        if stats.game_active:
            ship.update()
            gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
            gf.update_aliens(ai_settings, stats, screen, ship, aliens, bullets)

        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

In the main loop, we always need to call `check_events()`, even if the game is inactive. For example, we still need to know if the user presses Q to quit the game or clicks the button to close the window. We also continue updating the screen so we can make changes to the screen while waiting to see whether the player chooses to start a new game. The rest of the function calls only need to happen when the game is active, because when the game is inactive, we don't need to update the positions of game elements.

Now when you play Alien Invasion, the game should freeze when you've used up all of your ships.

TRY IT YOURSELF

13-6. Game Over: Using your code from Exercise 13-5 (page 284), keep track of the number of times the player misses the ball. When they've missed the ball three times, end the game.

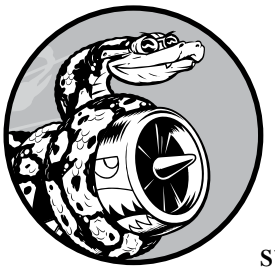
Summary

In this chapter you learned how to add a large number of identical elements to a game by creating a fleet of aliens. You learned how to use nested loops to create a grid of elements, and you made a large set of game elements move by calling each element's `update()` method. You learned to control the direction of objects on the screen and how to respond to events, such as when the fleet reaches the edge of the screen. You also learned how to detect and respond to collisions when bullets hit aliens and aliens hit the ship. Finally, you learned how to track the statistics in a game and use a `game_active` flag to determine when the game was over.

In the final chapter of this project, we'll add a Play button so the player can choose when to start their first game and whether to play again when the game ends. We'll speed up the game each time the player shoots down the entire fleet, and we'll add a scoring system. The final result will be a fully playable game!

14

SCORING



In this chapter we'll finish the Alien Invasion game. We'll add a Play button to start a game on demand or to restart a game once it ends. We'll also change the game so it speeds up when the player moves up a level, and we'll implement a scoring system. By the end of the chapter, you'll know enough to start writing games that increase in difficulty as a player progresses and that show scores.

Adding the Play Button

In this section we'll add a Play button that appears before a game begins and reappears when the game ends so the player can play again.

Right now the game begins as soon as you run *alien_invasion.py*. Let's start the game in an inactive state and then prompt the player to click a Play button to begin. To do this, enter the following in *game_stats.py*:

game_stats.py

```
    def __init__(self, ai_settings):
        """Initialize statistics."""
        self.ai_settings = ai_settings
        self.reset_stats()

        # Start game in an inactive state.
        self.game_active = False

    def reset_stats(self):
        --snip--
```

Now the game should start in an inactive state with no way for the player to start it until we make a Play button.

Creating a Button Class

Because Pygame doesn't have a built-in method for making buttons, we'll write a `Button` class to create a filled rectangle with a label. You can use this code to make any button in a game. Here's the first part of the `Button` class; save it as *button.py*:

button.py

```
import pygame.font

class Button():

❶   def __init__(self, ai_settings, screen, msg):
        """Initialize button attributes."""
        self.screen = screen
        self.screen_rect = screen.get_rect()

        # Set the dimensions and properties of the button.
❷       self.width, self.height = 200, 50
        self.button_color = (0, 255, 0)
        self.text_color = (255, 255, 255)
❸       self.font = pygame.font.SysFont(None, 48)

        # Build the button's rect object and center it.
❹       self.rect = pygame.Rect(0, 0, self.width, self.height)
        self.rect.center = self.screen_rect.center

        # The button message needs to be prepped only once.
❺       self.prep_msg(msg)
```

First we import the `pygame.font` module, which lets Pygame render text to the screen. The `__init__()` method takes the parameters `self`, the `ai_settings` and `screen` objects, and `msg`, which contains the text for the button ❶. We set the button dimensions at ❷, and then we set `button_color` to color the button's `rect` object bright green and set `text_color` to render the text in white.

At ❸ we prepare a `font` attribute for rendering text. The `None` argument tells Pygame to use the default font, and `48` determines the size of the text. To center the button on the screen, we create a `rect` for the button ❹ and set its `center` attribute to match that of the screen.

Pygame works with text by rendering the string you want to display as an image. At ❺ we call `prep_msg()` to handle this rendering.

Here's the code for `prep_msg()`:

button.py

```
    def prep_msg(self, msg):
        """Turn msg into a rendered image and center text on the button."""
❶       self.msg_image = self.font.render(msg, True, self.text_color,
            self.button_color)
❷       self.msg_image_rect = self.msg_image.get_rect()
        self.msg_image_rect.center = self.rect.center
```

The `prep_msg()` method needs a `self` parameter and the text to be rendered as an image (`msg`). The call to `font.render()` turns the text stored in `msg` into an image, which we then store in `msg_image` ❶. The `font.render()` method also takes a Boolean value to turn antialiasing on or off (antialiasing makes the edges of the text smoother). The remaining arguments are the specified font color and background color. We set antialiasing to `True` and set the text background to the same color as the button. (If you don't include a background color, Pygame will try to render the font with a transparent background.)

At ❷ we center the text image on the button by creating a `rect` from the image and setting its `center` attribute to match that of the button.

Finally, we create a `draw_button()` method that we can call to display the button onscreen:

button.py

```
    def draw_button(self):
        # Draw blank button and then draw message.
        self.screen.fill(self.button_color, self.rect)
        self.screen.blit(self.msg_image, self.msg_image_rect)
```

We call `screen.fill()` to draw the rectangular portion of the button. Then we call `screen.blit()` to draw the text image to the screen, passing it an image and the `rect` object associated with the image. This completes the `Button` class.

Drawing the Button to the Screen

We'll use the `Button` class to create a Play button. Because we need only one Play button, we'll create the button directly in *alien_invasion.py* as shown here:

alien_invasion.py

```
--snip--
from game_stats import GameStats
from button import Button
--snip--

def run_game():
    --snip--
    pygame.display.set_caption("Alien Invasion")

    # Make the Play button.
❶   play_button = Button(ai_settings, screen, "Play")
    --snip--

    # Start the main loop for the game.
    while True:
        --snip--
❷       gf.update_screen(ai_settings, screen, stats, ship, aliens, bullets,
            play_button)

run_game()
```

We import `Button` and create an instance called `play_button` ❶, and then we pass `play_button` to `update_screen()` so the button appears when the screen updates ❷.

Next, modify `update_screen()` so the Play button appears only when the game is inactive:

game_functions.py

```
def update_screen(ai_settings, screen, stats, ship, aliens, bullets,
        play_button):
    """Update images on the screen and flip to the new screen."""
    --snip--

    # Draw the play button if the game is inactive.
    if not stats.game_active:
        play_button.draw_button()

    # Make the most recently drawn screen visible.
    pygame.display.flip()
```

To make the Play button visible above all other elements on the screen, we draw it after all other game elements have been drawn and before flipping to a new screen. Now when you run Alien Invasion you should see a Play button in the center of the screen, as shown in Figure 14-1.

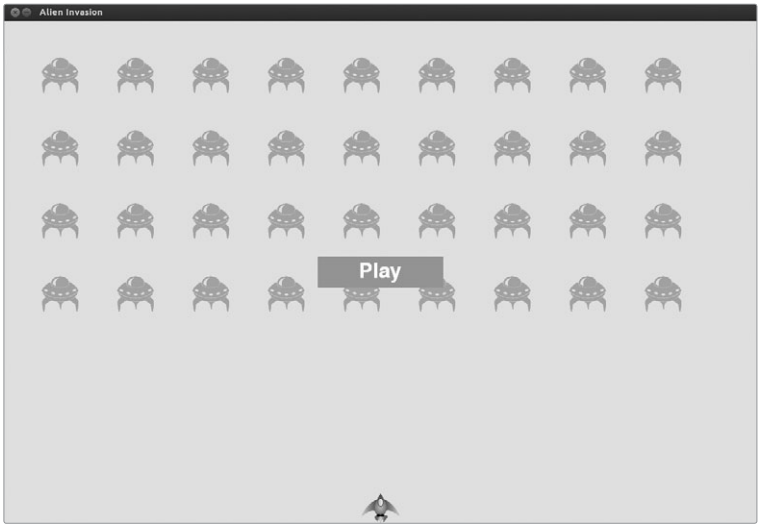


Figure 14-1: A Play button appears when the game is inactive.

Starting the Game

To start a new game when the player clicks Play, add the following code to *game_functions.py* to monitor mouse events over the button:

game_functions.py

```
def check_events(ai_settings, screen, stats, play_button, ship, bullets):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            --snip--
❶       elif event.type == pygame.MOUSEBUTTONDOWN:
❷           mouse_x, mouse_y = pygame.mouse.get_pos()
❸           check_play_button(stats, play_button, mouse_x, mouse_y)

def check_play_button(stats, play_button, mouse_x, mouse_y):
    """Start a new game when the player clicks Play."""
❹   if play_button.rect.collidepoint(mouse_x, mouse_y):
        stats.game_active = True
```

We've updated the definition of `check_events()` to accept the `stats` and `play_button` parameters. We'll use `stats` to access the `game_active` flag and `play_button` to check whether the Play button has been clicked.

Pygame detects a `MOUSEBUTTONDOWN` event when the player clicks anywhere on the screen ❶, but we want to restrict our game to respond to mouse clicks only on the Play button. To accomplish this, we use `pygame.mouse.get_pos()`, which returns a tuple containing the x- and y-coordinates of the mouse cursor when the mouse button is clicked ❷. We send these values to the function `check_play_button()` ❸, which uses `collidepoint()` to see if the point of the mouse click overlaps the region defined by the Play button's `rect` ❹. If so, we set `game_active` to `True`, and the game begins!

The call to `check_events()` in *alien_invasion.py* needs to pass two additional arguments, `stats` and `play_button`:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, stats, play_button, ship,
            bullets)
        --snip--
```

At this point, you should be able to start and play a full game. When the game ends, the value of `game_active` should become `False` and the Play button should reappear.

Resetting the Game

The code we just wrote works the first time the player clicks Play but not once the first game ends, because the conditions that caused the game to end haven't been reset.

To reset the game each time the player clicks Play, we need to reset the game statistics, clear out the old aliens and bullets, build a new fleet, and center the ship, as shown here:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship, aliens,
        bullets, mouse_x, mouse_y):
    """Start a new game when the player clicks Play."""
    if play_button.rect.collidepoint(mouse_x, mouse_y):
        # Reset the game statistics.
❶       stats.reset_stats()
        stats.game_active = True

        # Empty the list of aliens and bullets.
❷       aliens.empty()
        bullets.empty()

        # Create a new fleet and center the ship.
❸       create_fleet(ai_settings, screen, ship, aliens)
        ship.center_ship()
```

We update the definition of `check_play_button()` so it has access to `ai_settings`, `stats`, `ship`, `aliens`, and `bullets`. It needs these objects to reset the settings that have changed during the game and to refresh the visual elements of the game.

At ❶ we reset the game statistics, which gives the player three new ships. Then we set `game_active` to `True` (so the game will begin as soon as the code in this function finishes running), empty the `aliens` and `bullets` groups ❷, and create a new fleet and center the ship ❸.

The definition of `check_events()` needs to be modified, as does the call to `check_play_button()`:

game_functions.py

```
def check_events(ai_settings, screen, stats, play_button, ship, aliens,
        bullets):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            --snip--
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_x, mouse_y = pygame.mouse.get_pos()
❶           check_play_button(ai_settings, screen, stats, play_button, ship,
                aliens, bullets, mouse_x, mouse_y)
```

The definition of `check_events()` needs the `aliens` parameter, which it will pass to `check_play_button()`. We then update the call to `check_play_button()` so it passes the appropriate arguments ❶.

Now update the call to `check_events()` in *alien_invasion.py* so it passes the `aliens` argument:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, stats, play_button, ship,
            aliens, bullets)
        --snip--
```

The game will now reset properly each time you click Play, allowing you to play it as many times as you want!

Deactivating the Play Button

One issue with our Play button is that the button region on the screen will continue to respond to clicks even when the Play button isn't visible. Click the Play button area by accident once a game has begun and the game will restart!

To fix this, set the game to start only when `game_active` is `False`:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship, aliens,
        bullets, mouse_x, mouse_y):
    """Start a new game when the player clicks Play."""
❶   button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
❷   if button_clicked and not stats.game_active:
        # Reset the game statistics.
        --snip--
```

The flag `button_clicked` stores a `True` or `False` value ❶, and the game will restart only if Play is clicked *and* the game is not currently active ❷. To test this behavior, start a new game and repeatedly click where the Play button should be. If everything works as expected, clicking the Play button area should have no effect on the gameplay.

Hiding the Mouse Cursor

We want the mouse cursor visible in order to begin play, but once play begins it only gets in the way. To fix this, we'll make it invisible once the game becomes active:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship, aliens,
        bullets, mouse_x, mouse_y):
    """Start new game when the player clicks Play."""
    button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
    if button_clicked and not stats.game_active:
        # Hide the mouse cursor.
        pygame.mouse.set_visible(False)
        --snip--
```

Passing `False` to `set_visible()` tells Pygame to hide the cursor when the mouse is over the game window.

We'll make the cursor reappear once the game ends so the player can click Play to begin a new game. Here's the code to do that:

game_functions.py

```
def ship_hit(ai_settings, screen, stats, ship, aliens, bullets):
    """Respond to ship being hit by alien."""
    if stats.ships_left > 0:
        --snip--
    else:
        stats.game_active = False
        pygame.mouse.set_visible(True)
```

We make the cursor visible again as soon as the game becomes inactive, which happens in `ship_hit()`. Attention to details like this makes your game seem more professional and allows the player to focus on playing rather than figuring out the user interface.

TRY IT YOURSELF

14-1. Press P to Play: Because Alien Invasion uses keyboard input to control the ship, it's best to start the game with a keypress. Add code that lets the player press P to start. It may help to move some code from `check_play_button()` to a `start_game()` function that can be called from both `check_play_button()` and `check_keydown_events()`.

14-2. Target Practice: Create a rectangle at the right edge of the screen that moves up and down at a steady rate. Then have a ship appear on the left side of the screen that the player can move up and down while firing bullets at the moving, rectangular target. Add a Play button that starts the game, and when the player misses the target three times, end the game and make the Play button reappear. Let the player restart the game with this Play button.

Leveling Up

In our current game, once a player shoots down the entire alien fleet, the player reaches a new level, but the game difficulty doesn't change. Let's liven things up a bit and make the game more challenging by increasing the speed of the game each time a player clears the screen.

Modifying the Speed Settings

We'll first reorganize the `Settings` class to group the game settings into static and changing ones. We'll also make sure that settings that change over the course of a game reset when we start a new game. Here's the `__init__()` method for *settings.py*:

settings.py

```
    def __init__(self):
        """Initialize the game's static settings."""
        # Screen settings
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)

        # Ship settings
        self.ship_limit = 3

        # Bullet settings
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
        self.bullets_allowed = 3

        # Alien settings
        self.fleet_drop_speed = 10

        # How quickly the game speeds up
❶       self.speedup_scale = 1.1

❷       self.initialize_dynamic_settings()
```

We continue to initialize the settings that stay constant in the `__init__()` method. At ❶ we add a `speedup_scale` setting to control how quickly the game speeds up: a value of 2 will double the game speed every time the player reaches a new level; a value of 1 will keep the speed constant. A speed value like 1.1 should increase the speed enough to make the game challenging but not impossible. Finally, we call `initialize_dynamic_settings()` to initialize the values for attributes that need to change throughout the course of a game ❷.

Here's the code for `initialize_dynamic_settings()`:

settings.py

```
    def initialize_dynamic_settings(self):
        """Initialize settings that change throughout the game."""
        self.ship_speed_factor = 1.5
        self.bullet_speed_factor = 3
```

```
        self.alien_speed_factor = 1

        # fleet_direction of 1 represents right; -1 represents left.
        self.fleet_direction = 1
```

This method sets the initial values for the ship, bullet, and alien speeds. We'll increase these speeds as the player progresses in the game and reset them each time the player starts a new game. We include `fleet_direction` in this method so the aliens always move right at the beginning of a new game. To increase the speeds of the ship, bullets, and aliens each time the player reaches a new level, use `increase_speed()`:

settings.py

```
    def increase_speed(self):
        """Increase speed settings."""
        self.ship_speed_factor *= self.speedup_scale
        self.bullet_speed_factor *= self.speedup_scale
        self.alien_speed_factor *= self.speedup_scale
```

To increase the speed of these game elements, we multiply each speed setting by the value of `speedup_scale`.

We increase the game's tempo by calling `increase_speed()` in `check_bullet_alien_collisions()` when the last alien in a fleet has been shot down but before creating a new fleet:

game_functions.py

```
def check_bullet_alien_collisions(ai_settings, screen, ship, aliens, bullets):
    --snip--
    if len(aliens) == 0:
        # Destroy existing bullets, speed up game, and create new fleet.
        bullets.empty()
        ai_settings.increase_speed()
        create_fleet(ai_settings, screen, ship, aliens)
```

Changing the values of the speed settings `ship_speed_factor`, `alien_speed_factor`, and `bullet_speed_factor` is enough to speed up the entire game!

Resetting the Speed

We need to return any changed settings to their initial values each time the player starts a new game, or each new game would start with the increased speed settings of the previous game:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship, aliens,
        bullets, mouse_x, mouse_y):
    """Start a new game when the player clicks Play."""
    button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
    if button_clicked and not stats.game_active:
        # Reset the game settings.
        ai_settings.initialize_dynamic_settings()

        # Hide the mouse cursor.
        pygame.mouse.set_visible(False)
        --snip--
```

Playing Alien Invasion should be more fun and challenging now. Each time you clear the screen, the game should speed up and become slightly more difficult. If the game becomes too difficult too quickly, decrease the value of `settings.speedup_scale`, or if the game isn't challenging enough, increase the value slightly. Find a sweet spot by ramping up the difficulty in a reasonable amount of time. The first couple of screens should be easy, the next few challenging but doable, and subsequent screens almost impossibly difficult.

TRY IT YOURSELF

14-3. Challenging Target Practice: Start with your work from Exercise 14-2 (page 298). Make the target move faster as the game progresses, and restart at the original speed when the player clicks Play.

Scoring

Let's implement a scoring system to track the game's score in real time, as well as to display the high score, level, and the number of ships remaining.

The score is a game statistic, so we'll add a `score` attribute to `GameStats`:

game_stats.py

```
class GameStats():
    --snip--
    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.ai_settings.ship_limit
        self.score = 0
```

To reset the score each time a new game starts, we initialize `score` in `reset_stats()` rather than `__init__()`.

Displaying the Score

To display the score on the screen, we first create a new class, `Scoreboard`. For now this class will just display the current score, but we'll use it to report the high score, level, and number of ships remaining as well. Here's the first part of the class; save it as *scoreboard.py*:

scoreboard.py

```
import pygame.font

class Scoreboard():
    """A class to report scoring information."""

❶   def __init__(self, ai_settings, screen, stats):
        """Initialize scorekeeping attributes."""
        self.screen = screen
```

```
        self.screen_rect = screen.get_rect()
        self.ai_settings = ai_settings
        self.stats = stats

        # Font settings for scoring information.
❷       self.text_color = (30, 30, 30)
❸       self.font = pygame.font.SysFont(None, 48)

        # Prepare the initial score image.
❹       self.prep_score()
```

Because `Scoreboard` writes text to the screen, we begin by importing the `pygame.font` module. Next, we give `__init__()` the parameters `ai_settings`, `screen`, and `stats` so it can report the values we're tracking ❶. Then, we set a text color ❷ and instantiate a font object ❸.

To turn the text to be displayed into an image, we call `prep_score()` ❹, which we define here:

scoreboard.py

```
    def prep_score(self):
        """Turn the score into a rendered image."""
❶       score_str = str(self.stats.score)
❷       self.score_image = self.font.render(score_str, True, self.text_color,
            self.ai_settings.bg_color)

        # Display the score at the top right of the screen.
❸       self.score_rect = self.score_image.get_rect()
❹       self.score_rect.right = self.screen_rect.right - 20
❺       self.score_rect.top = 20
```

In `prep_score()`, we first turn the numerical value `stats.score` into a string ❶, and then pass this string to `render()`, which creates the image ❷. To display the score clearly onscreen, we pass the screen's background color to `render()` as well as a text color.

We'll position the score in the upper-right corner of the screen and have it expand to the left as the score increases and the width of the number grows. To make sure the score always lines up with the right side of the screen, we create a `rect` called `score_rect` ❸ and set its right edge 20 pixels from the right screen edge ❹. We then place the top edge 20 pixels down from the top of the screen ❺.

Finally, we create a `show_score()` method to display the rendered score image:

scoreboard.py

```
    def show_score(self):
        """Draw score to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
```

This method draws the score image to the screen at the location specified by `score_rect`.

Making a Scoreboard

To display the score, we'll create a `Scoreboard` instance in *alien_invasion.py*:

alien_invasion.py

```
--snip--
from game_stats import GameStats
from scoreboard import Scoreboard
--snip--
def run_game():
    --snip--
    # Create an instance to store game statistics and create a scoreboard.
    stats = GameStats(ai_settings)
❶   sb = Scoreboard(ai_settings, screen, stats)
    --snip--
    # Start the main loop for the game.
    while True:
        --snip--
❷       gf.update_screen(ai_settings, screen, stats, sb, ship, aliens,
            bullets, play_button)

run_game()
```

We import the new `Scoreboard` class and make an instance called `sb` after creating the `stats` instance ❶. We then pass `sb` to `update_screen()` so the score can be drawn to the screen ❷.

To display the score, modify `update_screen()` like this:

game_functions.py

```
def update_screen(ai_settings, screen, stats, sb, ship, aliens, bullets,
        play_button):
    --snip--
    # Draw the score information.
    sb.show_score()

    # Draw the play button if the game is inactive.
    if not stats.game_active:
        play_button.draw_button()

    # Make the most recently drawn screen visible.
    pygame.display.flip()
```

We add `sb` to the list of parameters that define `update_screen()` and call `show_score()` just before the Play button is drawn.

When you run Alien Invasion now, you should see 0 at the top right of the screen. (For now we just want to make sure that the score appears in the right place before developing the scoring system further.) Figure 14-2 shows the score as it appears before the game starts.

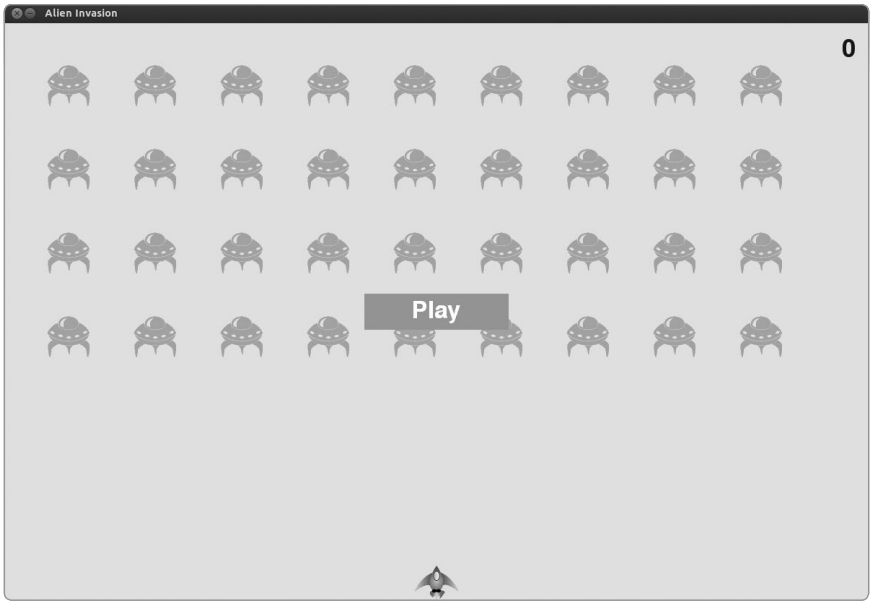


Figure 14-2: The score appears at the top-right corner of the screen.

Now to assign point values to each alien!

Updating the Score as Aliens Are Shot Down

To write a live score to the screen, we update the value of `stats.score` whenever an alien is hit, and then call `prep_score()` to update the score image. But first, let's determine how many points a player gets each time they shoot down an alien:

settings.py

```
    def initialize_dynamic_settings(self):
        --snip--

        # Scoring
        self.alien_points = 50
```

We'll increase the point value of each alien as the game progresses. To make sure this point value is reset each time a new game starts, we set the value in `initialize_dynamic_settings()`.

Update the score each time an alien is shot down in `check_bullet_alien_collisions()`:

game_functions.py

```
def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets):
    """Respond to bullet-alien collisions."""
    # Remove any bullets and aliens that have collided.
    collisions = pygame.sprite.groupcollide(bullets, aliens, True, True)
```

```
    if collisions:
❶       stats.score += ai_settings.alien_points
        sb.prep_score()
    --snip--
```

We update the definition of `check_bullet_alien_collisions()` to include the `stats` and `sb` parameters so it can update the score and the scoreboard. When a bullet hits an alien, Pygame returns a `collisions` dictionary. We check whether the dictionary exists, and if it does, the alien's value is added to the score ❶. We then call `prep_score()` to create a new image for the updated score.

We need to modify `update_bullets()` to make sure the appropriate arguments are passed between functions:

game_functions.py

```
def update_bullets(ai_settings, screen, stats, sb, ship, aliens, bullets):
    """Update position of bullets and get rid of old bullets."""
    --snip--

    check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets)
```

The definition of `update_bullets()` needs the additional parameters `stats` and `sb`. The call to `check_bullet_alien_collisions()` needs to include the `stats` and `sb` arguments as well.

We also need to modify the call to `update_bullets()` in the main `while` loop:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, stats, play_button, ship,
            aliens, bullets)
        if stats.game_active:
            ship.update()
            gf.update_bullets(ai_settings, screen, stats, sb, ship, aliens,
                bullets)
            --snip--
```

The call to `update_bullets()` needs the `stats` and `sb` arguments.

Now when you play Alien Invasion, you should be able to rack up points!

Making Sure to Score All Hits

As currently written, our code could miss some aliens. For example, if two bullets collide with aliens during the same pass through the loop or if we make an extra wide bullet to hit multiple aliens, the player will receive points only for one of the aliens killed. To fix this, let's refine the way that alien bullet collisions are detected.

In `check_bullet_alien_collisions()`, any bullet that collides with an alien becomes a key in the `collisions` dictionary. The value associated with each bullet is a list of aliens it has collided with. We loop through the `collisions` dictionary to make sure we award points for each alien hit:

game_functions.py

```
def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets):
    --snip--
    if collisions:
❶       for aliens in collisions.values():
            stats.score += ai_settings.alien_points * len(aliens)
            sb.prep_score()
    --snip--
```

If the `collisions` dictionary has been defined, we loop through all values in the `collisions` dictionary. Remember that each value is a list of aliens hit by a single bullet. We multiply the value of each alien by the number of aliens in each list and add this amount to the current score. To test this, change the width of a bullet to 300 pixels and verify that you receive points for each alien you hit with your extra wide bullets; then return the bullet width to normal.

Increasing Point Values

Because the game gets more difficult each time a player reaches a new level, aliens in later levels should be worth more points. To implement this functionality, we'll add code to increase the point value when the game's speed increases:

settings.py

```
class Settings():
    """A class to store all settings for Alien Invasion."""

    def __init__(self):
        --snip--
        # How quickly the game speeds up
        self.speedup_scale = 1.1
        # How quickly the alien point values increase
❶       self.score_scale = 1.5

        self.initialize_dynamic_settings()

    def increase_speed(self):
        """Increase speed settings and alien point values."""
        self.ship_speed_factor *= self.speedup_scale
        self.bullet_speed_factor *= self.speedup_scale
        self.alien_speed_factor *= self.speedup_scale

❷       self.alien_points = int(self.alien_points * self.score_scale)
```

We define a rate at which points increase, which we call `score_scale` ❶. A small increase in speed (1.1) makes the game grow challenging quickly,

but in order to see a notable difference in scoring you need to change the alien point value by a larger amount (1.5). Now when we increase the speed of the game, we also increase the point value of each hit ❷. We use the `int()` function to increase the point value by whole integers.

To see the value of each alien, add a `print` statement to the method `increase_speed()` in `Settings`:

settings.py

```
    def increase_speed(self):
        --snip--
        self.alien_points = int(self.alien_points * self.score_scale)
        print(self.alien_points)
```

You should see the new point value in the terminal every time you reach a new level.

NOTE *Be sure to remove the `print` statement after verifying that the point value is increasing, or it may affect the performance of your game and distract the player.*

Rounding the Score

Most arcade-style shooting games report scores as multiples of 10, so let's follow that lead with our scoring. Let's also format the score to include comma separators in large numbers. We'll make this change in `Scoreboard`:

scoreboard.py

```
    def prep_score(self):
        """Turn the score into a rendered image."""
❶       rounded_score = int(round(self.stats.score, -1))
❷       score_str = "{:,}".format(rounded_score)
        self.score_image = self.font.render(score_str, True, self.text_color,
            self.ai_settings.bg_color)
        --snip--
```

The `round()` function normally rounds a decimal number to a set number of decimal places given as the second argument. However, if you pass a negative number as the second argument, `round()` will round the value to the nearest 10, 100, 1000, and so on. The code at ❶ tells Python to round the value of `stats.score` to the nearest 10 and store it in `rounded_score`.

NOTE *In Python 2.7, `round()` always returns a decimal value, so we use `int()` to make sure the score is reported as an integer. If you're using Python 3, you can leave out the call to `int()`.*

At ❷, a string formatting directive tells Python to insert commas into numbers when converting a numerical value to a string—for example, to output `1,000,000` instead of `1000000`. Now when you run the game, you should see a neatly formatted, rounded score even when you rack up lots of points, as shown in Figure 14-3.

Figure 14-3: Rounded score with comma separators

High Scores

Every player wants to beat a game's high score, so let's track and report high scores to give players something to work toward. We'll store high scores in `GameStats`:

game_stats.py

```
    def __init__(self, ai_settings):
        --snip--
        # High score should never be reset.
        self.high_score = 0
```

Because the high score should never be reset, we initialize `high_score` in `__init__()` rather than in `reset_stats()`.

Now we'll modify `Scoreboard` to display the high score. Let's start with the `__init__()` method:

scoreboard.py

```
    def __init__(self, ai_settings, screen, stats):
        --snip--
        # Prepare the initial score images.
        self.prep_score()
❶       self.prep_high_score()
```

The high score will be displayed separately from the score, so we need a new method, `prep_high_score()`, to prepare the high score image ❶.

Here's the `prep_high_score()` method:

scoreboard.py

```
    def prep_high_score(self):
        """Turn the high score into a rendered image."""
❶       high_score = int(round(self.stats.high_score, -1))
```

```
❷          high_score_str = "{:,}".format(high_score)
❸          self.high_score_image = self.font.render(high_score_str, True,
               self.text_color, self.ai_settings.bg_color)

           # Center the high score at the top of the screen.
           self.high_score_rect = self.high_score_image.get_rect()
❹          self.high_score_rect.centerx = self.screen_rect.centerx
❺          self.high_score_rect.top = self.score_rect.top
```

We round the high score to the nearest 10 ❶ and format it with commas ❷. We then generate an image from the high score ❸, center the high score `rect` horizontally ❹, and set its `top` attribute to match the top of the score image ❺.

The `show_score()` method now draws the current score at the top right and the high score at the top center of the screen:

scoreboard.py

```
    def show_score(self):
        """Draw the score to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
```

To check for high scores, we'll write a new function, `check_high_score()`, in *game_functions.py*:

game_functions.py

```
def check_high_score(stats, sb):
    """Check to see if there's a new high score."""
❶   if stats.score > stats.high_score:
        stats.high_score = stats.score
        sb.prep_high_score()
```

The function `check_high_score()` takes two parameters, `stats` and `sb`. It uses `stats` to check the current score and the high score, and it needs `sb` to modify the high score image when necessary. At ❶ we check the current score against the high score. If the current score is greater, we update the value of `high_score` and call `prep_high_score()` to update the image of the high score.

We need to call `check_high_score()` each time an alien is hit after updating the score in `check_bullet_alien_collisions()`:

game_functions.py

```
def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets):
    --snip--
    if collisions:
        for aliens in collisions.values():
            stats.score += ai_settings.alien_points * len(aliens)
            sb.prep_score()
        check_high_score(stats, sb)
    --snip--
```

We call `check_high_score()` when the `collisions` dictionary is present, and we do so after updating the score for all the aliens that have been hit.

The first time you play Alien Invasion your score will be the high score, so it will be displayed as both the current and high score. But when you start a second game, your high score should appear in the middle and your current score at the right, as shown in Figure 14-4.

Figure 14-4: The high score is shown at the top center of the screen.

Displaying the Level

To display the player's level in the game, we first need an attribute in `GameStats` representing the current level. To reset the level at the start of each new game, initialize it in `reset_stats()`:

game_stats.py

```
    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.ai_settings.ship_limit
        self.score = 0
        self.level = 1
```

To have `Scoreboard` display the current level (just below the current score), we call a new method, `prep_level()`, from `__init__()`:

scoreboard.py

```
    def __init__(self, ai_settings, screen, stats):
        --snip--

        # Prepare the initial score images.
        self.prep_score()
        self.prep_high_score()
        self.prep_level()
```

Here's `prep_level()`:

scoreboard.py

```
    def prep_level(self):
        """Turn the level into a rendered image."""
❶       self.level_image = self.font.render(str(self.stats.level), True,
                self.text_color, self.ai_settings.bg_color)

        # Position the level below the score.
        self.level_rect = self.level_image.get_rect()
❷       self.level_rect.right = self.score_rect.right
❸       self.level_rect.top = self.score_rect.bottom + 10
```

The method `prep_level()` creates an image from the value stored in `stats.level` ❶ and sets the image's `right` attribute to match the score's `right` attribute ❷. It then sets the `top` attribute 10 pixels beneath the bottom of the score image to leave space between the score and the level ❸.

We also need to update `show_score()`:

scoreboard.py

```
    def show_score(self):
        """Draw scores and ships to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
        self.screen.blit(self.level_image, self.level_rect)
```

This adds a line to draw the level image to the screen.

We'll increment `stats.level` and update the level image in `check_bullet_alien_collisions()`:

game_functions.py

```
def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets):
    --snip--
    if len(aliens) == 0:
        # If the entire fleet is destroyed, start a new level.
        bullets.empty()
        ai_settings.increase_speed()

        # Increase level.
❶       stats.level += 1
❷       sb.prep_level()

        create_fleet(ai_settings, screen, ship, aliens)
```

If a fleet is destroyed, we increment the value of `stats.level` ❶ and call `prep_level()` to make sure the new level is displayed correctly ❷.

To make sure the scoring and level images are updated properly at the start of a new game, trigger a reset when the Play button is clicked:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, sb, play_button, ship,
        aliens, bullets, mouse_x, mouse_y):
    """Start a new game when the player clicks Play."""
    button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
    if button_clicked and not stats.game_active:
```

```
        --snip--

        # Reset the game statistics.
        stats.reset_stats()
        stats.game_active = True

        # Reset the scoreboard images.
❶       sb.prep_score()
        sb.prep_high_score()
        sb.prep_level()

        # Empty the list of aliens and bullets.
        aliens.empty()
        bullets.empty()

        --snip--
```

The definition of `check_play_button()` needs the `sb` object. To reset the scoreboard images, we call `prep_score()`, `prep_high_score()`, and `prep_level()` after resetting the relevant game settings ❶.

Now pass `sb` from `check_events()` so `check_play_button()` has access to the scoreboard object:

game_functions.py

```
def check_events(ai_settings, screen, stats, sb, play_button, ship, aliens,
        bullets):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            --snip--
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_x, mouse_y = pygame.mouse.get_pos()
❶           check_play_button(ai_settings, screen, stats, sb, play_button,
                ship, aliens, bullets, mouse_x, mouse_y)
```

The definition of `check_events()` needs `sb` as a parameter, so the call to `check_play_button()` can include `sb` as an argument ❶.

Finally, update the call to `check_events()` in *alien_invasion.py* so it passes `sb` as well:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        gf.check_events(ai_settings, screen, stats, sb, play_button, ship,
            aliens, bullets)
        --snip--
```

Now you can see how many levels you've completed, as shown in Figure 14-5.

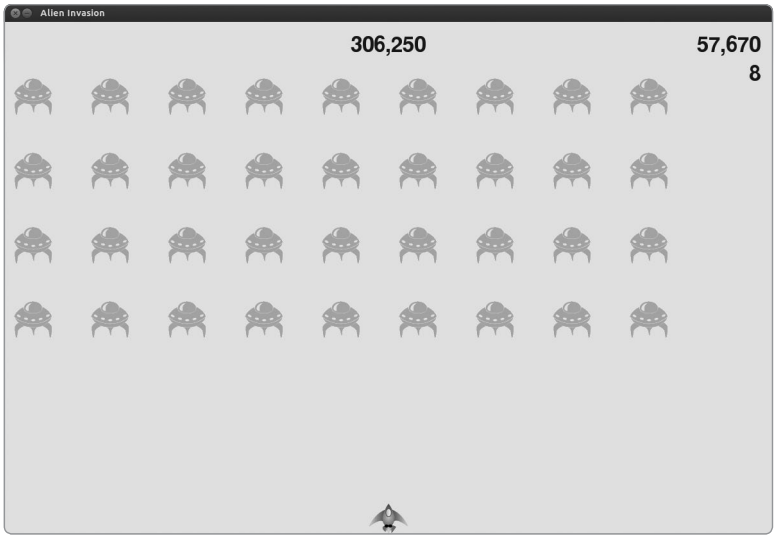


Figure 14-5: The current level is reported just below the current score.

NOTE *In some classic games, the scores have labels, such as Score, High Score, and Level. We've omitted these labels because the meaning of each number becomes clear once you've played the game. To include these labels, add them to the score strings just before the calls to* `font.render()` *in* `Scoreboard`*.*

Displaying the Number of Ships

Finally, let's display the number of ships the player has left, but this time, let's use a graphic. To do so, we'll draw ships in the upper-left corner of the screen to represent how many ships are left, like many classic arcade games do.

First, we need to make `Ship` inherit from `Sprite` so we can create a group of ships:

ship.py

```
import pygame
from pygame.sprite import Sprite

❶ class Ship(Sprite):

    def __init__(self, ai_settings, screen):
        """Initialize the ship and set its starting position."""
❷       super(Ship, self).__init__()
        --snip--
```

Here we import `Sprite`, make sure `Ship` inherits from `Sprite` ❶, and call `super()` at the beginning of `__init__()` ❷.

Next, we need to modify `Scoreboard` to create a group of ships we can display. Here's the `import` statements and `__init__()`:

scoreboard.py

```
import pygame.font
from pygame.sprite import Group

from ship import Ship

class Scoreboard():
    """A class to report scoring information."""

    def __init__(self, ai_settings, screen, stats):
        --snip--
        self.prep_level()
        self.prep_ships()
       --snip--
```

Because we're making a group of ships, we import the `Group` and `Ship` classes. We call `prep_ships()` after the call to `prep_level()`.

Here's `prep_ships()`:

scoreboard.py

```
    def prep_ships(self):
        """Show how many ships are left."""
❶       self.ships = Group()
❷       for ship_number in range(self.stats.ships_left):
            ship = Ship(self.ai_settings, self.screen)
❸           ship.rect.x = 10 + ship_number * ship.rect.width
❹           ship.rect.y = 10
❺           self.ships.add(ship)
```

The `prep_ships()` method creates an empty group, `self.ships`, to hold the ship instances ❶. To fill this group, a loop runs once for every ship the player has left ❷. Inside the loop we create a new ship and set each ship's x-coordinate value so the ships appear next to each other with a 10-pixel margin on the left side of the group of ships ❸. We set the y-coordinate value 10 pixels down from the top of the screen so the ships line up with the score image ❹. Finally, we add each new ship to the group `ships` ❺.

Now we need to draw the ships to the screen:

scoreboard.py

```
    def show_score(self):
        --snip--
        self.screen.blit(self.level_image, self.level_rect)
        # Draw ships.
        self.ships.draw(self.screen)
```

To display the ships on the screen, we call `draw()` on the group, and Pygame draws each ship.

To show the player how many ships they have to start with, we call `prep_ships()` when a new game starts. We do this in `check_play_button()` in *game_functions.py*:

game_functions.py

```
def check_play_button(ai_settings, screen, stats, sb, play_button, ship,
        aliens, bullets, mouse_x, mouse_y):
    """Start a new game when the player clicks Play."""
    button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
    if button_clicked and not stats.game_active:
        --snip--
        # Reset the scoreboard images.
        sb.prep_score()
        sb.prep_high_score()
        sb.prep_level()
        sb.prep_ships()
        --snip--
```

We also call `prep_ships()` when a ship is hit to update the display of ship images when the player loses a ship:

game_functions.py

```
❶ def update_aliens(ai_settings, screen, stats, sb, ship, aliens, bullets):
    --snip--
    # Look for alien-ship collisions.
    if pygame.sprite.spritecollideany(ship, aliens):
❷       ship_hit(ai_settings, screen, stats, sb, ship, aliens, bullets)

    # Look for aliens hitting the bottom of the screen.
❸   check_aliens_bottom(ai_settings, screen, stats, sb, ship, aliens, bullets)

❹ def ship_hit(ai_settings, screen, stats, sb, ship, aliens, bullets):
    """Respond to ship being hit by alien."""
    if stats.ships_left > 0:
        # Decrement ships_left.
        stats.ships_left -= 1

        # Update scoreboard.
❺       sb.prep_ships()

        # Empty the list of aliens and bullets.
        --snip--
```

We first add the parameter `sb` to the definition of `update_aliens()` ❶. We then pass `sb` to `ship_hit()` ❷ and `check_aliens_bottom()` so each has access to the scoreboard object ❸.

Then we update the definition of `ship_hit()` to include `sb` ❹. We call `prep_ships()` after decreasing the value of `ships_left` ❺, so the correct number of ships is displayed each time a ship is destroyed.

There's a call to `ship_hit()` in `check_aliens_bottom()`, so update that function as well:

game_functions.py

```
def check_aliens_bottom(ai_settings, screen, stats, sb, ship, aliens,
        bullets):
    """Check if any aliens have reached the bottom of the screen."""
    screen_rect = screen.get_rect()
    for alien in aliens.sprites():
        if alien.rect.bottom >= screen_rect.bottom:
            # Treat this the same as if a ship got hit.
            ship_hit(ai_settings, screen, stats, sb, ship, aliens, bullets)
            break
```

Now `check_aliens_bottom()` accepts `sb` as a parameter, and we add an `sb` argument in the call to `ship_hit()`.

Finally, pass `sb` in the call to `update_aliens()` in *alien_invasion.py*:

alien_invasion.py

```
    # Start the main loop for the game.
    while True:
        --snip--
        if stats.game_active:
            ship.update()
            gf.update_bullets(ai_settings, screen, stats, sb, ship, aliens,
                bullets)
            gf.update_aliens(ai_settings, screen, stats, sb, ship, aliens,
                bullets)
            --snip--
```

Figure 14-6 shows the complete scoring system with the remaining ships displayed at the top left of the screen.

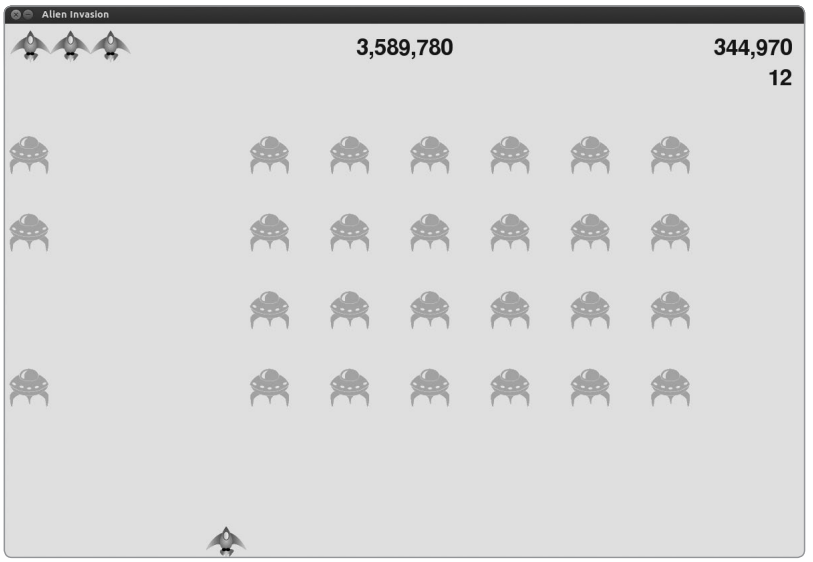


Figure 14-6: The complete scoring system for Alien Invasion

TRY IT YOURSELF

14-4. All-Time High Score: The high score is reset every time a player closes and restarts Alien Invasion. Fix this by writing the high score to a file before calling `sys.exit()` and reading the high score in when initializing its value in `GameStats`.

14-5. Refactoring: Look for functions and methods that are doing more than one task, and refactor them to keep your code organized and efficient. For example, move some of the code in `check_bullet_alien_collisions()`, which starts a new level when the fleet of aliens has been destroyed, to a function called `start_new_level()`. Also, move the four separate method calls in the `__init__()` method in `Scoreboard` to a method called `prep_images()` to shorten `__init__()`. The `prep_images()` method could also help `check_play_button()` or `start_game()` if you've already refactored `check_play_button()`.

NOTE *Before attempting to refactor the project, see Appendix D to learn how to restore the project to a working state if you introduce bugs while refactoring.*

14-6. Expanding Alien Invasion: Think of a way to expand Alien Invasion. For example, you could program the aliens to shoot bullets down at the ship or add shields for your ship to hide behind, which can be destroyed by bullets from either side. Or use something like the `pygame.mixer` module to add sound effects like explosions and shooting sounds.

Summary

In this chapter you learned to build a Play button to start a new game and how to detect mouse events and hide the cursor in active games. You can use what you've learned to create other buttons in your games, like a Help button to display instructions on how to play. You also learned how to modify the speed of a game as it progresses, how to implement a progressive scoring system, and how to display information in textual and nontextual ways.