PART II

PROJECTS

Congratulations! You now know enough about Python to start building interactive and meaningful projects. Creating your own projects will teach you new skills and solidify your understanding of the concepts introduced in Part I.

Part II contains three types of projects, and you can choose to do any or all of these projects in whichever order you like. Here's a brief description of each project to help you decide which to dig into first.

Alien Invasion: Making a Game with Python

In the *Alien Invasion* project (Chapters 12, 13, and 14), you'll use the Pygame package to develop a 2D game in which the aim is to shoot down a fleet of aliens as they drop down the screen in levels that increase in speed and difficulty. At the end of the project, you'll have learned skills that will enable you to develop your own 2D games in Pygame.

Data Visualization

The Data Visualization project starts in Chapter 15, in which you'll learn to generate data and create a series of functional and beautiful visualizations of that data using Matplotlib and Plotly. Chapter 16 teaches you to access data from online sources and feed it into a visualization package to create plots of weather data and a map of global earthquake activity. Finally, Chapter 17 shows you how to write a program to automatically download

and visualize data. Learning to make visualizations allows you to explore the field of data mining, which is a highly sought-after skill in the world today.

Web Applications

In the Web Applications project (Chapters 18, 19, and 20), you'll use the Django package to create a simple web application that allows users to keep a journal about any number of topics they've been learning about. Users will create an account with a username and password, enter a topic, and then make entries about what they're learning. You'll also learn how to deploy your app so anyone in the world can access it.

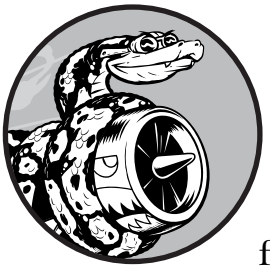
After completing this project, you'll be able to start building your own simple web applications, and you'll be ready to delve into more thorough resources on building applications with Django.

PROJECT 1

ALIEN INVASION

12

A SHIP THAT FIRES BULLETS



Let's build a game called *Alien Invasion*! 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 focus on the higher-level logic of game dynamics.

In this chapter, you'll set up Pygame, and then create a rocket 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 refine the game by setting limits on the number of ships you can use and adding a scoreboard.

While building this game, you'll also learn how to manage large projects that span multiple files. We'll refactor a lot of code and manage file contents to organize the project and make the code efficient.

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

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

Also, if you feel comfortable using version control, you might want to use it for this project. If you haven't used version control before, see Appendix D for an overview.

Planning Your Project

When you're building a large project, it's important to prepare a plan before you begin to write 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 general gameplay. Although the following 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 rocket 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 development phase, we'll make a ship that can move right and left and fires bullets when the player presses the spacebar. After setting up this behavior, we can create the aliens and refine the gameplay.

Installing Pygame

Before you begin coding, install Pygame. The `pip` module helps you download and install Python packages. To install Pygame, enter the following command at a terminal prompt:

```
$ python -m pip install --user pygame
```

This command tells Python to run the `pip` module and install the `pygame` package to the current user's Python installation. If you use a command

other than `python` to run programs or start a terminal session, such as `python3`, your command will look like this:

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

NOTE *If this command doesn't work on macOS, try running the command again without the `--user` flag.*

Starting the Game Project

We'll begin building the game by creating an empty Pygame window. Later, we'll draw the game elements, such as the ship and the aliens, on this window. We'll also make our game respond to user input, set the background color, and load a ship image.

Creating a Pygame Window and Responding to User Input

We'll make an empty Pygame window by creating a class to represent the game. In your text editor, create a new file and save it as *alien_invasion.py*; then enter the following:

alien_invasion.py

```
import sys

import pygame

class AlienInvasion:
    """Overall class to manage game assets and behavior."""

    def __init__(self):
        """Initialize the game, and create game resources."""
❶       pygame.init()

❷       self.screen = pygame.display.set_mode((1200, 800))
        pygame.display.set_caption("Alien Invasion")

    def run_game(self):
        """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()

if __name__ == '__main__':
    # Make a game instance, and run the game.
    ai = AlienInvasion()
    ai.run_game()
```

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

Alien Invasion starts as a class called `AlienInvasion`. In the `__init__()` method, the `pygame.init()` function initializes the background settings that Pygame needs to work properly ❶. At ❷, we call `pygame.display.set_mode()` to create a display window, on which we'll draw all the game's graphical elements. The argument `(1200, 800)` is a tuple that defines the dimensions of the game window, which will be 1200 pixels wide by 800 pixels high. (You can adjust these values depending on your display size.) We assign this display window to the attribute `self.screen`, so it will be available in all methods in the class.

The object we assigned to `self.screen` is called a *surface*. A surface in Pygame is a part of the screen where a game element can be displayed. Each element in the game, like an alien or a ship, is its own surface. The surface returned by `display.set_mode()` represents the entire game window. When we activate the game's animation loop, this surface will be redrawn on every pass through the loop, so it can be updated with any changes triggered by user input.

The game is controlled by the `run_game()` method. This method contains a `while` loop ❸ that runs continually. The `while` loop 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 write this *event loop* to *listen* for events and perform appropriate tasks depending on the kinds of events that occur. The `for` loop at ❹ is an event loop.

To access the events that Pygame detects, we'll use the `pygame.event.get()` function. This function returns a list of events that have taken place since the last time this function was called. Any keyboard or mouse event will cause this `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 simply draws an empty screen on each pass through the `while` loop, erasing the old screen so only the new screen is visible. When we move the game elements around, `pygame.display.flip()` continually updates the display to show the new positions of game elements and hides the old ones, creating the illusion of smooth movement.

At the end of the file, we create an instance of the game, and then call `run_game()`. We place `run_game()` in an `if` block that only runs if the file is called directly. When you run this *alien_invasion.py* file, 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. We'll do this at the end of the `__init__()` method.

alien_invasion.py

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

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

    def run_game(self):
        --snip--
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()

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

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

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 different RGB values to create up to 16 million colors. The color value (230, 230, 230) mixes equal amounts of red, blue, and green, which produces a light gray background color. We assign this color to `self.bg_color` ❶.

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

Creating a Settings Class

Each time we introduce new functionality into the game, we'll typically create 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 these values in one place. This approach allows us to work with just one settings object any time we need to access an individual setting. This also makes it easier to modify the game's appearance and behavior as our project grows: to modify the game, we'll simply change some values in *settings.py*, which we'll create next, instead of searching for different settings throughout the project.

Create a new file named *settings.py* inside your *alien_invasion* folder, and add this 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` in the project and use it to access our settings, we need to modify *alien_invasion.py* as follows:

alien_invasion.py

```
--snip--
import pygame

from settings import Settings

class AlienInvasion:
    """Overall class to manage game assets and behavior."""

    def __init__(self):
        """Initialize the game, and create game resources."""
        pygame.init()
❶       self.settings = Settings()

❷       self.screen = pygame.display.set_mode(
            (self.settings.screen_width, self.settings.screen_height))
        pygame.display.set_caption("Alien Invasion")

    def run_game(self):
            --snip--
            # Redraw the screen during each pass through the loop.
❸           self.screen.fill(self.settings.bg_color)

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

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

When you run *alien_invasion.py* now you won't yet see any changes, because all we've done is move the settings we were already using elsewhere. Now we're ready to start adding new elements to the screen.

Adding the Ship Image

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

When you're 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 use and modify, from a website like *https://pixabay.com/*.

You can use almost any type of image file in your game, but it's easiest when 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* or *.png* 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 or solid 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 at *https://nostarch.com/pythoncrashcourse2e/*. The file's background color matches the settings we're using in this project. Make a folder called *images* inside your main *alien_invasion* project folder. Save the file *ship.bmp* in the *images* folder.

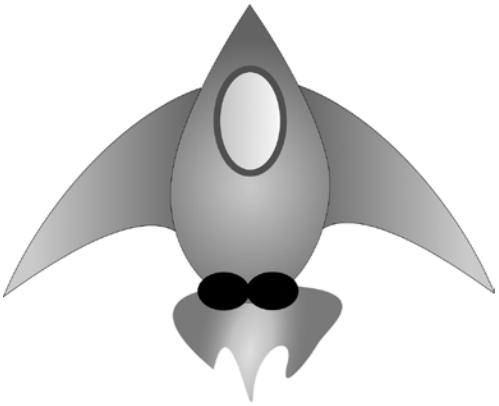


Figure 12-1: The ship for Alien Invasion

Creating the Ship Class

After choosing an image for the ship, we need to display it on the screen. To use our ship, we'll create a new `ship` module that will contain the class `Ship`. This class will manage most of the behavior of the player's ship:

ship.py

```
import pygame

class Ship:
    """A class to manage the ship."""

    def __init__(self, ai_game):
        """Initialize the ship and set its starting position."""
❶       self.screen = ai_game.screen
❷       self.screen_rect = ai_game.screen.get_rect()

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

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

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

Pygame is efficient because it lets you treat all 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. When Pygame needs to figure out whether two game elements have collided, for example, it can do this more quickly if it treats each object as a rectangle. 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. We'll treat the ship and the screen as rectangles in this class.

We import the `pygame` module before defining the class. The `__init__()` method of `Ship` takes two parameters: the `self` reference and a reference to the current instance of the `AlienInvasion` class. This will give `Ship` access to all the game resources defined in `AlienInvasion`. At ❶ we assign the screen to an attribute of `Ship`, so we can access it easily in all the methods in this class. At ❷ we access the screen's `rect` attribute using the `get_rect()` method and assign it to `self.screen_rect`. Doing so allows us to place the ship in the correct location on the screen.

To load the image, we call `pygame.image.load()` ❸ and give it the location of our ship image. This function returns a surface representing the ship, which we assign to `self.image`. When the image is loaded, we call `get_rect()` to access the ship surface's `rect` attribute so we can later use it to place the ship.

When you're 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, to place the object. You can set any of these values to establish 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. There are also attributes that combine these properties, such as `midbottom`, `midtop`, `midleft`, and `midright`. 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 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). These coordinates refer to the game window, not the physical screen.*

We'll position the ship at the bottom center of the screen. To do so, make the value of `self.rect.midbottom` match the `midbottom` attribute of the screen's rect ❹. Pygame uses 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 draws 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

class AlienInvasion:
    """Overall class to manage game assets and behavior."""

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

❶       self.ship = Ship(self)

    def run_game(self):
            --snip--
            # Redraw the screen during each pass through the loop.
            self.screen.fill(self.settings.bg_color)
❷           self.ship.blitme()

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

We import `Ship` and then make an instance of `Ship` after the screen has been created ❶. The call to `Ship()` requires one argument, an instance of `AlienInvasion`. The `self` argument here refers to the current instance of `AlienInvasion`. This is the parameter that gives `Ship` access to the game's resources, such as the `screen` object. We assign this `Ship` instance to `self.ship`.

After filling the background, we draw the ship on the screen by calling `ship.blitme()`, 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 the 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 _check_events() and _update_screen() Methods

In large 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 break the `run_game()` method, which is getting lengthy, into two helper methods. A *helper method* does work inside a class but isn't meant to be called through an instance. In Python, a single leading underscore indicates a helper method.

The _check_events() Method

We'll move the code that manages events to a separate method 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, such as updating the screen.

Here's the `AlienInvasion` class with the new `_check_events()` method, which only affects the code in `run_game()`:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
❶           self._check_events()
```

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

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

We make a new `_check_events()` method ❷ and move the lines that check whether the player has clicked to close the window into this new method.

To call a method from within a class, use dot notation with the variable `self` and the name of the method ❶. We call the method from inside the `while` loop in `run_game()`.

The _update_screen() Method

To further simplify `run_game()`, we'll move the code for updating the screen to a separate method called `_update_screen()`:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()
            self._update_screen()

    def _check_events(self):
        --snip--

    def _update_screen(self):
        """Update images on the screen, and flip to the new screen."""
        self.screen.fill(self.settings.bg_color)
        self.ship.blitme()

        pygame.display.flip()
```

We moved the code that draws the background and the ship and flips the screen to `_update_screen()`. Now the body of the main loop in `run_game()` is much simpler. It's easy to see that we're looking for new events and updating the screen on each pass through the loop.

If you've already built a number of games, you'll probably start out by breaking your code into methods like these. But if you've never tackled a project like this, you probably won't know how to structure your code. This approach of writing code that works and then restructuring it as it grows more complex gives you an idea of a realistic development process: you start out writing your code as simply as possible, and then refactor it as your project becomes more complex.

Now that we've restructured the code 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

Next, we'll give the player the ability to move the ship right and left. 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 we add this code, you'll learn how to control the movement of images on the screen and respond to user input.

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. We need to specify in our `_check_events()` method what kind of events we want the game to check for. Each keypress is registered as a `KEYDOWN` event.

When Pygame detects a `KEYDOWN` event, we need to check whether the key that was pressed is one that triggers a certain action. For example, if the player presses the right arrow key, we want to increase the ship's `rect.x` value to move the ship to the right:

alien_invasion.py

```
    def _check_events(self):
        """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                   self.ship.rect.x += 1
```

Inside `_check_events()` we add an `elif` block to the event loop to respond when Pygame detects a `KEYDOWN` event ❶. We check whether the key pressed, `event.key`, is the right arrow key ❷. The right arrow key is represented by `pygame.K_RIGHT`. If the right arrow key was pressed, we move the ship to the right by increasing the value of `self.ship.rect.x` by 1 ❸.

When you run *alien_invasion.py* now, the ship should 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 the 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 `moving_right` flag is `False`, the ship will be motionless. When the player presses the right arrow key, we'll set the flag to `True`, and when the player releases the key, 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 once on each pass through the `while` loop to update the position of the ship.

Here are the changes to `Ship`:

ship.py

```
class Ship:
    """A class to manage the ship."""

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

        # 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.x += 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. The `update()` method will be called through an instance of `Ship`, so it's not considered a helper method.

Now we need to 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:

alien_invasion.py

```
    def _check_events(self):
        """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                   self.ship.moving_right = True
❷           elif event.type == pygame.KEYUP:
```

```
                if event.key == pygame.K_RIGHT:
                    self.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`.

Next, we modify the `while` loop in `run_game()` so it calls the ship's `update()` method on each pass through the loop:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()
            self.ship.update()
            self._update_screen()
```

The ship's position will be updated 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 will be 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 straightforward. Again, we'll modify the `Ship` class and the `_check_events()` method. Here are the relevant changes to `__init__()` and `update()` in `Ship`:

ship.py

```
    def __init__(self, ai_game):
        --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.x += 1
        if self.moving_left:
            self.rect.x -= 1
```

In `__init__()`, we add a `self.moving_left` flag. In `update()`, we use two separate `if` blocks rather than an `elif` to allow the ship's `rect.x` value to be increased and then decreased when 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 right to left when the player might momentarily hold down both keys.

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

alien_invasion.py

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

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

When 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 ship's movement. Let's adjust the ship's speed and limit how far the ship can move so it can'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` 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 = 1.5
```

We set the initial value of `ship_speed` to `1.5`. When the ship moves now, its position is adjusted by 1.5 pixels rather than 1 pixel on each pass through the loop.

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 `x` store only integer values, so we need to make some modifications to `Ship`:

ship.py

```
class Ship:
    """A class to manage the ship."""

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

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

        # Store a decimal value for the ship's horizontal position.
❷       self.x = float(self.rect.x)

        # 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 x value, not the rect.
        if self.moving_right:
❸           self.x += self.settings.ship_speed
        if self.moving_left:
            self.x -= self.settings.ship_speed

        # Update rect object from self.x.
❹       self.rect.x = self.x

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

We create a `settings` attribute for `Ship`, so we can use it in `update()` ❶. Because we're adjusting the position of the ship by fractions of a pixel, we need to assign the position to a variable that can store a decimal value. You can use a decimal value to set an attribute of `rect`, but the `rect` will only keep the integer portion of that value. To keep track of the ship's position accurately, we define a new `self.x` attribute that can hold decimal values ❷. We use the `float()` function to convert the value of `self.rect.x` to a decimal and assign this value to `self.x`.

Now when we change the ship's position in `update()`, the value of `self.x` is adjusted by the amount stored in `settings.ship_speed` ❸. After `self.x` has been updated, we use the new value to update `self.rect.x`, which controls

the position of the ship ❹. Only the integer portion of `self.x` will be stored in `self.rect.x`, but that's fine for displaying the ship.

Now we can change the value of `ship_speed`, and any value greater than one will make the ship move faster. This will help make 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.

NOTE *If you're using macOS, you might notice that the ship moves very slowly, even with a high speed setting. You can remedy this problem by running the game in fullscreen mode, which we'll implement shortly.*

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 screen's edge. 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 x value, not the rect.
❶       if self.moving_right and self.rect.right < self.screen_rect.right:
            self.x += self.settings.ship_speed
❷       if self.moving_left and self.rect.left > 0:
            self.x -= self.settings.ship_speed

        # Update rect object from self.x.
        self.rect.x = self.x
```

This code checks the position of the ship before changing the value of `self.x`. The code `self.rect.right` returns the x-coordinate 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.x`.

When you run *alien_invasion.py* now, the ship should stop moving at either edge of the screen. This is pretty cool; all we've done is add a conditional test in an `if` statement, but it feels like the ship hits a wall or a force field at either edge of the screen!

Refactoring _check_events()

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

alien_invasion.py

```
    def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
```

```
            if event.type == pygame.QUIT:
                sys.exit()
            elif event.type == pygame.KEYDOWN:
                self._check_keydown_events(event)
            elif event.type == pygame.KEYUP:
                self._check_keyup_events(event)

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

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

We make two new helper methods: `_check_keydown_events()` and `_check_keyup_events()`. Each needs a `self` parameter and an `event` parameter. The bodies of these two methods are copied from `_check_events()`, and we've replaced the old code with calls to the new methods. The `_check_events()` method is simpler now with this cleaner code structure, which will make it easier to develop further responses to player input.

Pressing Q to Quit

Now that we're responding to keypresses efficiently, we can add another way to quit the game. It gets tedious to click the X at the top of the game window to end the game every time you test a new feature, so we'll add a keyboard shortcut to end the game when the player presses Q:

alien_invasion.py

```
    def _check_keydown_events(self, event):
        --snip--
        elif event.key == pygame.K_LEFT:
            self.ship.moving_left = True
        elif event.key == pygame.K_q:
            sys.exit()
```

In `_check_keydown_events()`, we add a new block that ends the game when the player presses Q. Now, when testing, you can press Q to close the game rather than using your cursor to close the window.

Running the Game in Fullscreen Mode

Pygame has a fullscreen mode that you might like better than running the game in a regular window. Some games look better in fullscreen mode, and macOS users might see better performance in fullscreen mode.

To run the game in fullscreen mode, make the following changes in `__init__()`:

alien_invasion.py

```
    def __init__(self):
        """Initialize the game, and create game resources."""
        pygame.init()
        self.settings = Settings()

❶       self.screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN)
❷       self.settings.screen_width = self.screen.get_rect().width
        self.settings.screen_height = self.screen.get_rect().height
        pygame.display.set_caption("Alien Invasion")
```

When creating the screen surface, we pass a size of `(0, 0)` and the parameter `pygame.FULLSCREEN` ❶. This tells Pygame to figure out a window size that will fill the screen. Because we don't know the width and height of the screen ahead of time, we update these settings after the screen is created ❷. We use the `width` and `height` attributes of the screen's rect to update the `settings` object.

If you like how the game looks or behaves in fullscreen mode, keep these settings. If you liked the game better in its own window, you can revert back to the original approach where we set a specific screen size for the game.

NOTE *Make sure you can quit by pressing Q before running the game in fullscreen mode; Pygame offers no default way to quit a game while in fullscreen mode.*

A Quick Recap

In the next section, we'll add the ability to shoot bullets, which involves adding a new file called *bullet.py* and making some modifications to some of the files we're already using. Right now, we have three files containing a number of classes 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*, contains the `AlienInvasion` class. This class creates a number of important attributes used throughout the game: the settings are assigned to `settings`, the main display surface is assigned to `screen`, and a `ship` instance is created in this file as well. The main loop of the game, a `while` loop, is also stored in this module. The `while` loop calls `_check_events()`, `ship.update()`, and `_update_screen()`.

The `_check_events()` method detects relevant events, such as keypresses and releases, and processes each of these types of events through the methods `_check_keydown_events()` and `_check_keyup_events()`. For now,

these methods manage the ship's movement. The `AlienInvasion` class also contains `_update_screen()`, which redraws the screen on each pass through the main loop.

The *alien_invasion.py* file is the only file you need to run when you want to play *Alien Invasion*. The other files—*settings.py* and *ship.py*—contain code that is imported 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.

ship.py

The *ship.py* file contains the `Ship` class. The `Ship` class 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 image of the ship is stored in *ship.bmp*, which is in the *images* folder.

TRY IT YOURSELF

12-3. Pygame Documentation: We're far enough into the game now that you might want to look at some of the Pygame documentation. The Pygame home page is at *https://www.pygame.org/*, and the home page for the documentation is at *https://www.pygame.org/docs/*. Just skim the documentation for now. You won't need it to complete this project, but it will help if you want to modify *Alien Invasion* or make your own game afterward.

12-4. 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.

12-5. 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, which is represented by 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

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

settings.py

```
    def __init__(self):
        --snip--
        # Bullet settings
        self.bullet_speed = 1.0
        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_game):
        """Create a bullet object at the ship's current position."""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        self.color = self.settings.bullet_color

        # Create a bullet rect at (0, 0) and then set correct position.
❶       self.rect = pygame.Rect(0, 0, self.settings.bullet_width,
            self.settings.bullet_height)
❷       self.rect.midtop = ai_game.ship.rect.midtop

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

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 current instance of `AlienInvasion`, and we call `super()` to inherit properly from `Sprite`. We also set attributes for the screen and settings objects, and for the bullet's color.

At ❶, we create the bullet's `rect` attribute. The bullet isn't 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 line, because the bullet's position depends on the ship's position. We get the width and height of the bullet from the values stored in self.settings.

At ❷, we set the bullet's midtop attribute to match the ship's midtop attribute. This will make the bullet emerge from the top of the ship, making it look like the bullet is fired from the ship. We store a decimal value for the bullet's y-coordinate so we can make fine adjustments to the bullet's speed ❸.

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.settings.bullet_speed
        # 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. To update the position, we subtract the amount stored in settings.bullet_speed from self.y ❶. We then use the value of self.y to set the value of self.rect.y ❷.

The bullet_speed setting allows us to increase the speed of the bullets as the game progresses or as needed to refine the game's behavior. Once a bullet is fired, we never change the value of its x-coordinate, so it will travel vertically in a straight line even if the ship moves.

When we want to draw a bullet, we 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. We'll create a group in AlienInvasion 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 pygame.sprite.Group class, 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.

We'll create the group in __init__():

alien_invasion.py

```
    def __init__(self):
        --snip--
        self.ship = Ship(self)
        self.bullets = pygame.sprite.Group()
```

Then we need to update the position of the bullets on each pass through the `while` loop:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()
            self.ship.update()
❶           self.bullets.update()
            self._update_screen()
```

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

Firing Bullets

In `AlienInvasion`, we need to modify `_check_keydown_events()` to fire a bullet when the player presses the spacebar. We don't need to change `_check_keyup_events()` because nothing happens when the spacebar is released. We also need to modify `_update_screen()` to make sure each bullet is drawn to the screen before we call `flip()`.

We know there will be a bit of work to do when we fire a bullet, so let's write a new method, `_fire_bullet()`, to handle this work:

alien_invasion.py

```
--snip--
from ship import Ship
❶ from bullet import Bullet

class AlienInvasion:
    --snip--
    def _check_keydown_events(self, event):
        --snip--
        elif event.key == pygame.K_q:
            sys.exit()
❷       elif event.key == pygame.K_SPACE:
            self._fire_bullet()

    def _check_keyup_events(self, event):
        --snip--

    def _fire_bullet(self):
        """Create a new bullet and add it to the bullets group."""
❸       new_bullet = Bullet(self)
❹       self.bullets.add(new_bullet)

    def _update_screen(self):
        """Update images on the screen, and flip to the new screen."""
        self.screen.fill(self.settings.bg_color)
        self.ship.blitme()
❺       for bullet in self.bullets.sprites():
            bullet.draw_bullet()
```

```
        pygame.display.flip()
--snip--
```

First, we import `Bullet` ❶. Then we call `_fire_bullet()` when the spacebar is pressed ❷. In `_fire_bullet()`, we make an instance of `Bullet` and call it `new_bullet` ❸. We then add it to the group `bullets` using the `add()` method ❹. The `add()` method is similar to `append()`, but it's a method that's written specifically for Pygame groups.

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.

When 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*.

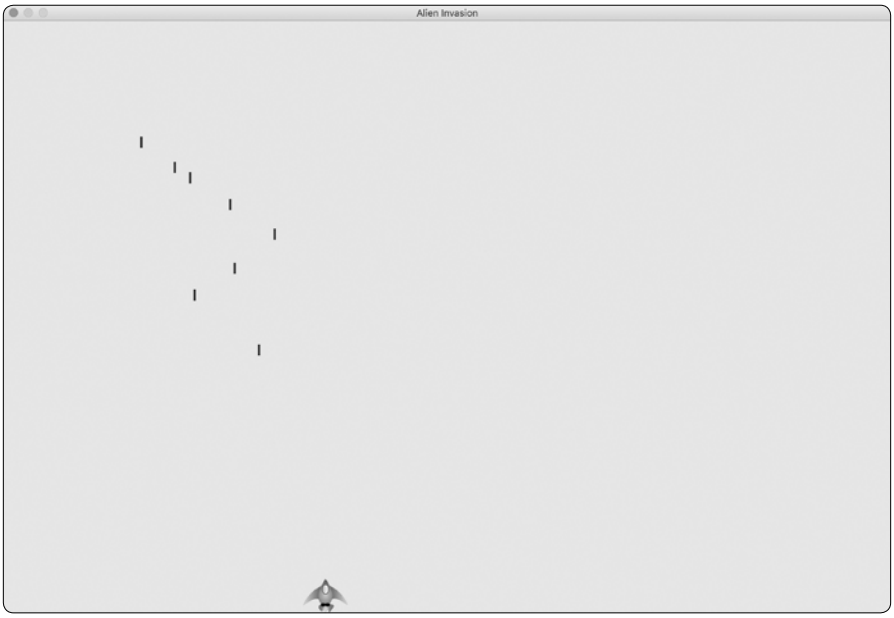


Figure 12-3: The ship after firing a series of bullets

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

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()
            self.ship.update()
            self.bullets.update()

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

            self._update_screen()
```

When you use a `for` loop with a list (or a group in Pygame), Python expects that the list will stay the same length as long as the loop is running. Because we can't remove items from a list or group within a `for` loop, 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()` call to show how many bullets currently exist in the game and verify that they're being deleted when they reach the top of the screen.

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 series of bullets has cleared the top of the screen. After you run the game and verify that bullets are being deleted properly, remove the `print()` call. 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; doing so encourages 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
        --snip--
        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 `AlienInvasion` to check how many bullets exist before creating a new bullet in `_fire_bullet()`:

alien_invasion.py

```
    def _fire_bullet(self):
        """Create a new bullet and add it to the bullets group."""
        if len(self.bullets) < self.settings.bullets_allowed:
            new_bullet = Bullet(self)
            self.bullets.add(new_bullet)
```

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

Creating the _update_bullets() Method

We want to keep the `AlienInvasion` class reasonably well organized, so now that we've written and checked the bullet management code, we can move it to a separate method. We'll create a new method called `_update_bullets()` and add it just before `_update_screen()`:

alien_invasion.py

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

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

The code for `_update_bullets()` is cut and pasted from `run_game()`; all we've done here is clarify the comments.

The `while` loop in `run_game()` looks simple again:

alien_invasion.py

```
        while True:
            self._check_events()
            self.ship.update()
            self._update_bullets()
            self._update_screen()
```

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

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

TRY IT YOURSELF

12-6. 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 and 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 you can adjust them more easily. You saw how to draw an image to the screen and give the player control over the movement of game elements. You created elements that move on their own, like bullets flying up a screen, and deleted objects that are no longer needed. You also 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 the chapter, 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*. We'll add one alien near the top of the screen and then 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 large 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: for example, you can

confine a character inside the walls of a maze or pass a ball between two characters. We'll 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 the Project

When you're beginning a new phase of development on a large 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'll:

- 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 entire 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 allotted number of ships.

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

You should also review your existing code when you 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 any cluttered or inefficient code. We've been refactoring as we go, so there isn't any code that we need to refactor at this point.

Creating the First Alien

Placing one alien on the screen is like placing a ship on the screen. Each alien's behavior 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 at *https://nostarch.com/pythoncrashcourse2e/*. This image has a gray background, which matches the screen's background color. Make sure you save the image file you choose in the *images* folder.

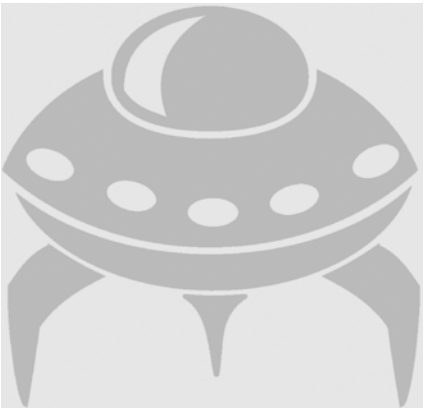


Figure 13-1: The alien we'll use to build the fleet

Creating the Alien Class

Now we'll write the `Alien` class and save it as *alien.py*:

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_game):
        """Initialize the alien and set its starting position."""
        super().__init__()
        self.screen = ai_game.screen

        # 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 horizontal position.
❷       self.x = float(self.rect.x)
```

Most of this class is like the `Ship` class except for the aliens' placement on the screen. We initially place each alien near the top-left corner of the screen; we add 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 so it's easy to see. We're mainly

concerned with the aliens' horizontal speed, so we'll track the horizontal position of each alien precisely ❷.

This `Alien` class doesn't need a method for drawing it to the screen; instead, we'll use a Pygame group method that automatically draws all the elements of a group to the screen.

Creating an Instance of the Alien

We want to create an instance of `Alien` so we can see the first alien on the screen. Because it's part of our setup work, we'll add the code for this instance at the end of the `__init__()` method in `AlienInvasion`. Eventually, we'll create an entire fleet of aliens, which will be quite a bit of work, so we'll make a new helper method called `_create_fleet()`.

The order of methods in a class doesn't matter, as long as there's some consistency to how they're placed. I'll place `_create_fleet()` just before the `_update_screen()` method, but anywhere in `AlienInvasion` will work. First, we'll import the `Alien` class.

Here are the updated `import` statements for *alien_invasion.py*:

alien_invasion.py

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

And here's the updated `__init__()` method:

alien_invasion.py

```
    def __init__(self):
        --snip--
        self.ship = Ship(self)
        self.bullets = pygame.sprite.Group()
        self.aliens = pygame.sprite.Group()

        self._create_fleet()
```

We create a group to hold the fleet of aliens, and we call `_create_fleet()`, which we're about to write.

Here's the new `_create_fleet()` method:

alien_invasion.py

```
    def _create_fleet(self):
        """Create the fleet of aliens."""
        # Make an alien.
        alien = Alien(self)
        self.aliens.add(alien)
```

In this method, we're creating one instance of `Alien`, and then adding it to the group that will hold the fleet. The alien will be placed in the default upper-left area of the screen, which is perfect for the first alien.

To make the alien appear, we need to call the group's `draw()` method in `_update_screen()`:

alien_invasion.py

```
    def _update_screen(self):
        --snip--
        for bullet in self.bullets.sprites():
            bullet.draw_bullet()
        self.aliens.draw(self.screen)

        pygame.display.flip()
```

When you call `draw()` on a group, Pygame draws each element in the group at the position defined by its `rect` attribute. The `draw()` method requires one argument: a surface on which to draw the elements from the group. Figure 13-2 shows the first alien on the screen.

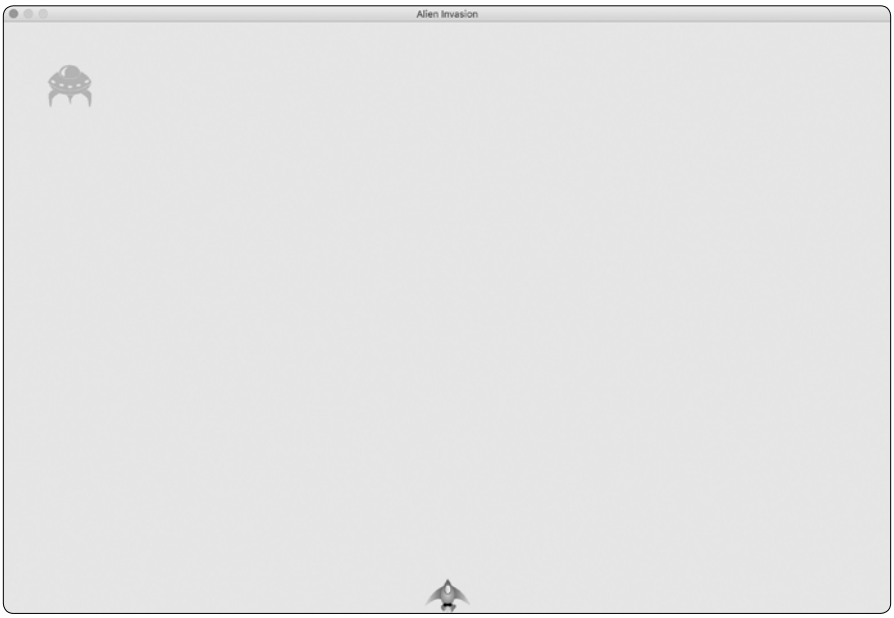


Figure 13-2: The first alien appears.

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 `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 = 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. We use *floor division* (`//`), which divides two numbers and drops any remainder, so we'll get an integer number of aliens:

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

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

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

Creating a Row of Aliens

We're ready to generate a full row of aliens. Because our code for making a single alien works, we'll rewrite `_create_fleet()` to make a whole row of aliens:

alien_invasion.py

```
    def _create_fleet(self):
        """Create the 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(self)
❷       alien_width = alien.rect.width
❸       available_space_x = self.settings.screen_width - (2 * alien_width)
        number_aliens_x = available_space_x // (2 * alien_width)

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

We've already thought through most of this code. We need to know the alien's width and height 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.

Next, we set up a loop that counts from 0 to the number of aliens we need to make ❹. In the main body of the loop, we create a new alien and then set its x-coordinate value to place it in the row ❺. 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. We use the alien's `x` attribute to set the position of its rect. Then we add each new alien to the group `aliens`.

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

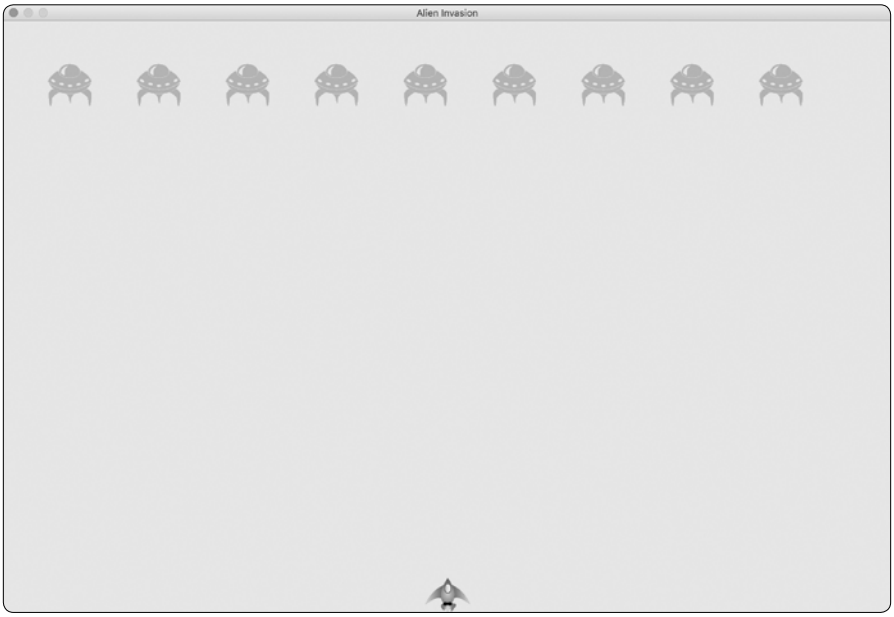


Figure 13-3: The first row of aliens

The first row is offset to the left, which is actually good for gameplay. The reason is that 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 might look slightly different on your system.*

Refactoring _create_fleet()

If the code we've written so far was all we need to create a fleet, we'd probably leave `_create_fleet()` as is. But we have more work to do, so let's clean up the method a bit. We'll add a new helper method, `_create_alien()`, and call it from `_create_fleet()`:

alien_invasion.py

```
    def _create_fleet(self):
        --snip--
        # Create the first row of aliens.
        for alien_number in range(number_aliens_x):
            self._create_alien(alien_number)

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

The method `_create_alien()` requires one parameter in addition to `self`: it needs the alien number that's currently being created. We use the same body we made for `_create_fleet()` except that we get the width of an alien inside the method instead of passing it as an argument. This refactoring will make it easier to add new rows and create an entire fleet.

Adding Rows

To finish the fleet, we'll determine the number of rows that fit on the screen and then repeat the loop for creating the aliens in one row until we have the correct number of rows. 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 = 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. We use floor division because we can only make an integer number of rows. (Again, if these calculations are off, we'll see it right away and adjust our approach 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:

alien_invasion.py

```
    def _create_fleet(self):
        --snip--
        alien = Alien(self)
❶       alien_width, alien_height = alien.rect.size
        available_space_x = self.settings.screen_width - (2 * alien_width)
        number_aliens_x = available_space_x // (2 * alien_width)

        # Determine the number of rows of aliens that fit on the screen.
        ship_height = self.ship.rect.height
❷       available_space_y = (self.settings.screen_height -
                             (3 * alien_height) - ship_height)
        number_rows = available_space_y // (2 * alien_height)

        # Create the full fleet of aliens.
❸       for row_number in range(number_rows):
            for alien_number in range(number_aliens_x):
                self._create_alien(alien_number, row_number)

    def _create_alien(self, alien_number, row_number):
        """Create an alien and place it in the row."""
        alien = Alien(self)
        alien_width, alien_height = alien.rect.size
        alien.x = alien_width + 2 * alien_width * alien_number
        alien.rect.x = alien.x
❹       alien.rect.y = alien.rect.height + 2 * alien.rect.height * row_number
        self.aliens.add(alien)
```

We need the width and height of an alien, so at ❶ we use the attribute `size`, which contains a tuple with the width and height of a `rect` object. To calculate the number of rows we can fit on the screen, we write our `available_space_y` calculation right after the calculation for `available_space_x` ❷. 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.

To create multiple rows, we use two nested loops: one outer and one inner loop ❸. The inner loop creates the aliens in one row. The outer loop counts from 0 to the number of rows we want; Python uses the code for making a single row and repeats 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 ❹ 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 previous 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.

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

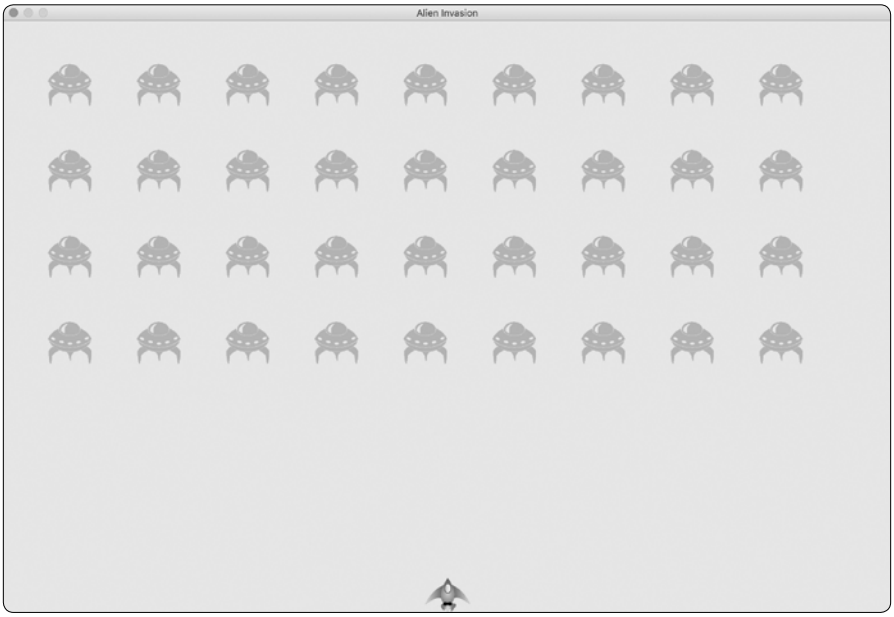


Figure 13-4: The full fleet appears.

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 the 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 = 1.0
```

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

alien.py

```
    def __init__(self, ai_game):
        """Initialize the alien and set its starting position."""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        --snip--

    def update(self):
        """Move the alien to the right."""
❶       self.x += self.settings.alien_speed
❷       self.rect.x = self.x
```

We create a settings parameter in `__init__()` so we can access the alien's speed in `update()`. Each time we update an alien's position, we move it to the right by the amount stored in `alien_speed`. 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 already have calls to update the ship and bullet positions. Now we'll add a call to update the position of each alien as well:

alien_invasion.py

```
        while True:
            self._check_events()
            self.ship.update()
            self._update_bullets()
            self._update_aliens()
            self._update_screen()
```

We're about to write some code to manage the movement of the fleet, so we create a new method called `_update_aliens()`. We set the aliens' positions to update after the bullets have been updated, because we'll soon be checking to see whether any bullets hit any aliens.

Where you place this method in the module is not critical. But to keep the code organized, I'll place it just after `_update_bullets()` to match the order of method calls in the `while` loop. Here's the first version of `_update_aliens()`:

alien_invasion.py

```
    def _update_aliens(self):
        """Update the positions of all aliens in the fleet."""
        self.aliens.update()
```

We use the `update()` method on the `aliens` group, which 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 = 1.0
        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 Whether an Alien Has Hit the Edge

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. This code is part of the `Alien` class:

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 or self.rect.left <= 0:
              return True

      def update(self):
          """Move the alien right or left."""
❷         self.x += (self.settings.alien_speed *
                         self.settings.fleet_direction)
          self.rect.x = self.x
```

We can call the new method `check_edges()` on any alien to see whether 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`. It's at the left edge if its `left` value is less than or equal to 0 ❶.

We modify the method `update()` to allow motion to the left or right by multiplying the alien's speed by the value of `fleet_direction` ❷. If `fleet_direction` is 1, the value of `alien_speed` 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. Therefore, we need to add some code to `AlienInvasion` because that's where we'll check whether any aliens are at the left or right edge. We'll make this happen by writing the methods `_check_fleet_edges()` and `_change_fleet_direction()`, and then modifying `_update_aliens()`. I'll put these new methods after `_create_alien()`, but again the placement of these methods in the class isn't critical.

alien_invasion.py

```
      def _check_fleet_edges(self):
          """Respond appropriately if any aliens have reached an edge."""
❶         for alien in self.aliens.sprites():
              if alien.check_edges():
❷                 self._change_fleet_direction()
                  break

      def _change_fleet_direction(self):
          """Drop the entire fleet and change the fleet's direction."""
          for alien in self.aliens.sprites():
❸             alien.rect.y += self.settings.fleet_drop_speed
          self.settings.fleet_direction *= -1
```

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 ❷. In `_change_fleet_direction()`, we loop through all the aliens and drop each one using the setting `fleet_drop_speed` ❸; then we change the value of `fleet_direction` by multiplying its current value by –1. The line that changes the fleet's direction isn't part of the `for` loop. We want to change each alien's vertical position, but we only want to change the direction of the fleet once.

Here are the changes to `_update_aliens()`:

alien_invasion.py

```
def _update_aliens(self):
    """
    Check if the fleet is at an edge,
      then update the positions of all aliens in the fleet.
    """
    self._check_fleet_edges()
    self.aliens.update()
```

We've modified the method by calling `_check_fleet_edges()` before updating each alien's position.

When 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 start 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 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 the position of all the bullets.

The `sprite.groupcollide()` function compares the rects of each element in one group with the rects of each element in another group. In this case, it 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 will be a bullet, and the corresponding value will be the alien that was hit. (We'll also use this dictionary when we implement a scoring system in Chapter 14.)

Add the following code to the end of `_update_bullets()` to check for collisions between bullets and aliens:

alien_invasion.py

```
    def _update_bullets(self):
        """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(
                self.bullets, self.aliens, True, True)
```

The new code we added compares the positions of all the bullets in `self.bullets` and all the aliens in `self.aliens`, and identifies any that overlap. 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 to delete the bullets and aliens that have collided. (To make a high-powered bullet that can 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.)

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

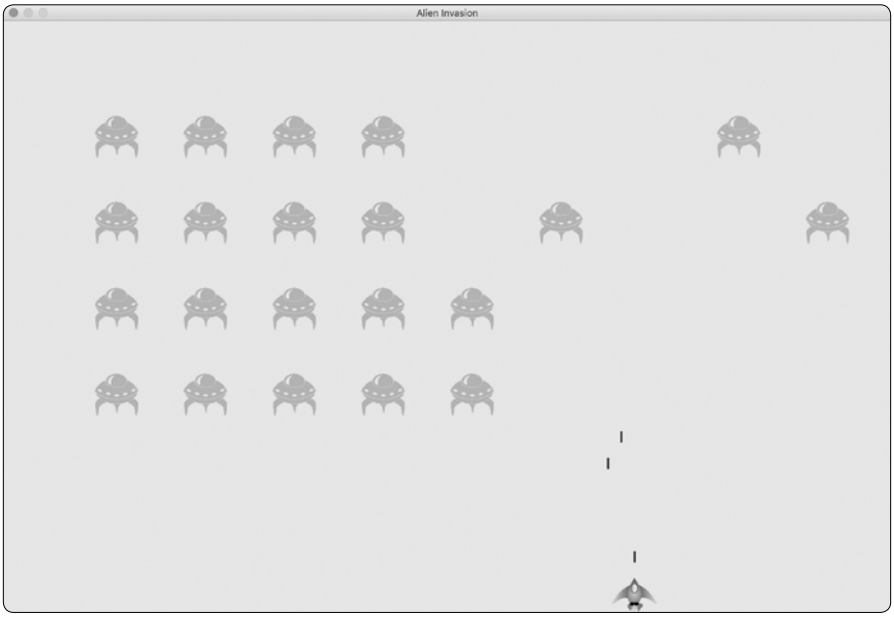


Figure 13-5: We can shoot aliens!

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 the game. For example, it's a lot of work to shoot down every alien on the screen multiple times to test whether 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 really wide bullets that remain active even after they've hit an alien (see Figure 13-6). Try setting `bullet_width` to 300, or even 3000, to see how quickly you can shoot down the fleet!

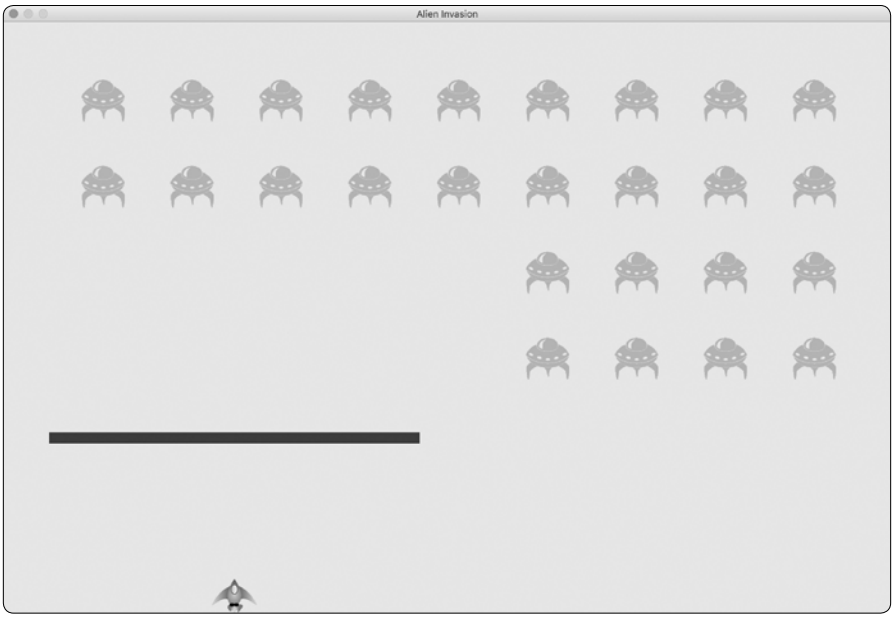


Figure 13-6: Extra-powerful bullets make some aspects of the game easier to test.

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 when you're finished testing a feature.

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, we first check whether the `aliens` group is empty. If it is, we make a call to `_create_fleet()`. We'll perform this check at the end of `_update_bullets()`, because that's where individual aliens are destroyed.

alien_invasion.py

```
    def _update_bullets(self):
        --snip--
❶       if not self.aliens:
            # Destroy existing bullets and create new fleet.
❷           self.bullets.empty()
            self._create_fleet()
```

At ❶, we check whether the `aliens` group is empty. An empty group evaluates to `False`, so this is a simple way to check whether the group 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.

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 might find that the bullets aren't traveling at the best speed for gameplay. They might be a little slow on your system or way too fast. At this point, you can modify the settings to make the gameplay interesting and enjoyable on your system.

We modify the speed of the bullets by adjusting the value of `bullet_speed` in *settings.py*. On my system, I'll adjust the value of `bullet_speed` to 1.5, so the bullets travel a little faster:

settings.py

```
        # Bullet settings
        self.bullet_speed = 1.5
        self.bullet_width = 3
        --snip--
```

The best value for this setting depends on your system's speed, so find a value that works for you. You can adjust other settings as well.

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 and alien collisions to a separate method:

alien_invasion.py

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

        self._check_bullet_alien_collisions()

    def _check_bullet_alien_collisions(self):
        """Respond to bullet-alien collisions."""
        # Remove any bullets and aliens that have collided.
```

```
        collisions = pygame.sprite.groupcollide(
                self.bullets, self.aliens, True, True)

        if not self.aliens:
            # Destroy existing bullets and create new fleet.
            self.bullets.empty()
            self._create_fleet()
```

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

TRY IT YOURSELF

13-5. Sideways Shooter Part 2: We've come a long way since Exercise 12-6, Sideways Shooter. For this exercise, try to develop Sideways Shooter to the same point we've brought *Alien Invasion* to. Add a fleet of aliens, and make them move sideways toward the ship. Or, write code that places aliens at random positions along the right side of the screen and then sends them toward the ship. Also, write code that makes the aliens disappear when they're hit.

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 when they make contact. 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. The game will end when the player has used up all their ships.

Detecting Alien and 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 and ship collisions immediately after updating the position of each alien in `AlienInvasion`:

alien_invasion.py

```
    def _update_aliens(self):
        --snip--
        self.aliens.update()

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

The `spritecollideany()` function takes two arguments: a sprite and a group. The function looks for any member of the group that has 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 has 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 and ship collisions works correctly. Writing a `print()` call is a simple way to ensure we're detecting these collisions properly.

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

Responding to Alien and Ship Collisions

Now we need to figure out exactly what will happen 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_game):
        """Initialize statistics."""
        self.settings = ai_game.settings
        self.reset_stats()

    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.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 `reset_stats()` method 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 should be stored in *settings.py* as `ship_limit`:

settings.py

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

We also need to make a few changes in *alien_invasion.py* to create an instance of `GameStats`. First, we'll update the `import` statements at the top of the file:

alien_invasion.py

```
import sys
from time import sleep

import pygame

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

We import the `sleep()` function from the `time` module in the Python standard library so we can pause the game for a moment when the ship is hit. We also import `GameStats`.

We'll create an instance of `GameStats` in `__init__()`:

alien_invasion.py

```
    def __init__(self):
        --snip--
        self.screen = pygame.display.set_mode(
            (self.settings.screen_width, self.settings.screen_height))
        pygame.display.set_caption("Alien Invasion")

        # Create an instance to store game statistics.
        self.stats = GameStats(self)

        self.ship = Ship(self)
        --snip--
```

We make the instance after creating the game window but before defining other game elements, such as the ship.

When an alien hits the ship, we'll 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 a new method called `_ship_hit()`. We'll call this method from `_update_aliens()` when an alien hits the ship:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to the ship being hit by an alien."""
```

```
            # Decrement ships_left.
❶           self.stats.ships_left -= 1

            # Get rid of any remaining aliens and bullets.
❷           self.aliens.empty()
            self.bullets.empty()

            # Create a new fleet and center the ship.
❸           self._create_fleet()
            self.ship.center_ship()

            # Pause.
❹           sleep(0.5)
```

The new method `_ship_hit()` coordinates the response when an alien hits a ship. Inside `_ship_hit()`, the number of ships left is reduced by 1 at ❶, after which we empty the groups `aliens` and `bullets` ❷.

Next, we create a new fleet and center the ship ❸. (We'll add the method `center_ship()` to `Ship` in a moment.) Then we add a 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 ❹. The `sleep()` call pauses program execution for half a second, long enough for the player to see that the alien has hit the ship. When the `sleep()` function ends, code execution moves on to the `_update_screen()` method, which draws the new fleet to the screen.

In `_update_aliens()`, we replace the `print()` call with a call to `_ship_hit()` when an alien hits the ship:

alien_invasion.py

```
    def _update_aliens(self):
        --snip--
        if pygame.sprite.spritecollideany(self.ship, self.aliens):
            self._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.rect.midbottom = self.screen_rect.midbottom
        self.x = float(self.rect.x)
```

We center the ship the same way we did in `__init__()`. After centering it, we reset the `self.x` attribute, which allows us to track the ship's exact position.

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 have the game respond the same way it does when an alien hits the ship. To check when this happens, add a new method in *alien_invasion.py*:

alien_invasion.py

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

The method `_check_aliens_bottom()` checks 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'll call this method from `_update_aliens()`:

alien_invasion.py

```
    def _update_aliens(self):
        --snip--
        # Look for alien-ship collisions.
        if pygame.sprite.spritecollideany(self.ship, self.aliens):
            self._ship_hit()

        # Look for aliens hitting the bottom of the screen.
        self._check_aliens_bottom()
```

We call `_check_aliens_bottom()` after updating the positions of all the aliens and after looking for alien and 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. We'll set this flag at the end of the `__init__()` method in `GameStats`:

game_stats.py

```
    def __init__(self, ai_game):
        --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` when the player has used up all their ships:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to ship being hit by alien."""
        if self.stats.ships_left > 0:
            # Decrement ships_left.
            self.stats.ships_left -= 1
           --snip--
            # Pause.
            sleep(0.5)
        else:
            self.stats.game_active = False
```

Most of `_ship_hit()` is unchanged. We've moved all 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

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

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()

            if self.stats.game_active:
                self.ship.update()
                self._update_bullets()
                self._update_aliens()

            self._update_screen()
```

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 your ships.

TRY IT YOURSELF

13-6. Game Over: In Sideways Shooter, keep track of the number of times the ship is hit and the number of times an alien is hit by the ship. Decide on an appropriate condition for ending the game, and stop the game when this situation occurs.

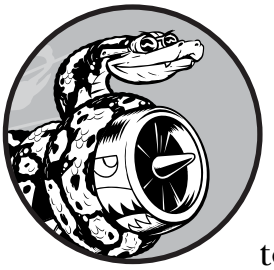
Summary

In this chapter, you learned how to add a large number of identical elements to a game by creating a fleet of aliens. You used 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 to respond to specific situations, such as when the fleet reaches the edge of the screen. You detected and responded to collisions when bullets hit aliens and aliens hit the ship. You also learned how to track statistics in a game and use a `game_active` flag to determine when the game is over.

In the next and 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 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 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, modify the `__init__()` method of `GameStats`:

game_stats.py

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

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

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_game, msg):
        """Initialize button attributes."""
        self.screen = ai_game.screen
        self.screen_rect = self.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_game`

object, and `msg`, which contains the button's text ❶. We set the button dimensions at ❷, and then 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` specifies 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 `self.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 in `AlienInvasion`. First, we'll update the `import` statements:

alien_invasion.py

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

Because we need only one Play button, we'll create the button in the `__init__()` method of `AlienInvasion`. We can place this code at the very end of `__init__()`:

alien_invasion.py

```
    def __init__(self):
        --snip--
        self._create_fleet()

        # Make the Play button.
        self.play_button = Button(self, "Play")
```

This code creates an instance of `Button` with the label *Play*, but it doesn't draw the button to the screen. We'll call the button's `draw_button()` method in `_update_screen()`:

alien_invasion.py

```
    def _update_screen(self):
        --snip--
        self.aliens.draw(self.screen)

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

        pygame.display.flip()
```

To make the Play button visible above all other elements on the screen, we draw it after all the other elements have been drawn but before flipping to a new screen. We include it in an `if` block, so the button only appears when the game is inactive.

Now when you run *Alien Invasion*, you should see a Play button in the center of the screen, as shown in Figure 14-1.

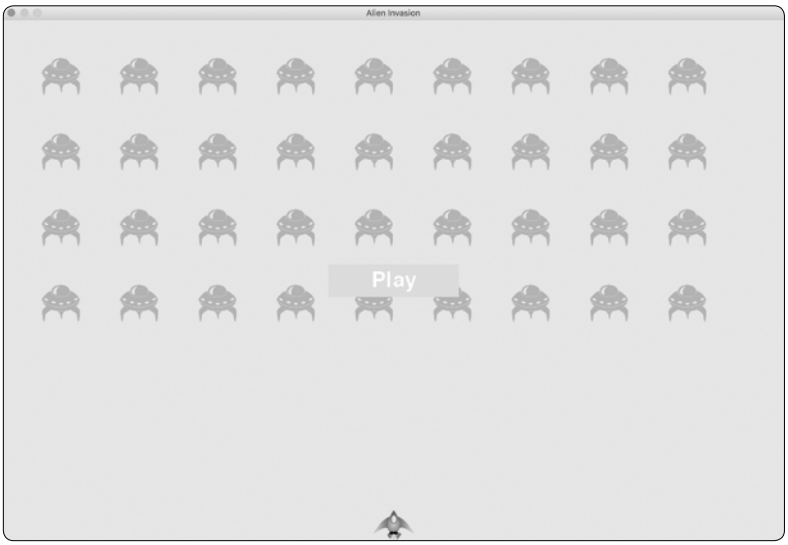


Figure 14-1: A Play button appears when the game is inactive.

Starting the Game

To start a new game when the player clicks Play, add the following `elif` block to the end of `_check_events()` to monitor mouse events over the button:

alien_invasion.py

```
    def _check_events(self):
        """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_pos = pygame.mouse.get_pos()
❸               self._check_play_button(mouse_pos)
```

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 mouse cursor's x- and y-coordinates when the mouse button is clicked ❷. We send these values to the new method `_check_play_button()` ❸.

Here's `_check_play_button()`, which I chose to place after `_check_events()`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
❶       if self.play_button.rect.collidepoint(mouse_pos):
            self.stats.game_active = True
```

We use the `rect` method `collidepoint()` to check whether the point of the mouse click overlaps the region defined by the Play button's `rect` ❶. If so, we set `game_active` to `True`, and the game begins!

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 Play button code we just wrote works the first time the player clicks Play. But it doesn't work after 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:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
        if self.play_button.rect.collidepoint(mouse_pos):
            # Reset the game statistics.
❶           self.stats.reset_stats()
            self.stats.game_active = True

            # Get rid of any remaining aliens and bullets.
❷           self.aliens.empty()
```

```
            self.bullets.empty()

            # Create a new fleet and center the ship.
❸           self._create_fleet()
            self.ship.center_ship()
```

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. We empty the `aliens` and `bullets` groups ❷, and then create a new fleet and center the ship ❸.

Now the game will 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. If you click the Play button area by accident after a game begins, the game will restart!

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

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
❶       button_clicked = self.play_button.rect.collidepoint(mouse_pos)
❷       if button_clicked and not self.stats.game_active:
            # Reset the game statistics.
            self.stats.reset_stats()
            --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 to be visible to begin play, but once play begins, it just gets in the way. To fix this, we'll make it invisible when the game becomes active. We can do this at the end of the `if` block in `_check_play_button()`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
        button_clicked = self.play_button.rect.collidepoint(mouse_pos)
        if button_clicked and not self.stats.game_active:
            --snip--
            # Hide the mouse cursor.
            pygame.mouse.set_visible(False)
```

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 again to begin a new game. Here's the code to do that:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to ship being hit by alien."""
        if self.stats.ships_left > 0:
            --snip--
        else:
            self.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 more professional looking 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 would be useful to start the game with a keypress. Add code that lets the player press P to start. It might help to move some code from `_check_play_button()` to a `_start_game()` method that can be called from `_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 game's speed 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

during the 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 those 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 value like 1.1 should increase the speed enough to make the game challenging but not impossible. Finally, we call the `initialize_dynamic_settings()` method to initialize the values for attributes that need to change throughout the 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 = 1.5
        self.bullet_speed = 3.0
        self.alien_speed = 1.0

        # 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. We don't need to increase the value of `fleet_drop_speed`, because when the aliens move faster across the screen, they'll also come down the screen faster.

To increase the speeds of the ship, bullets, and aliens each time the player reaches a new level, we'll write a new method called `increase_speed()`:

settings.py

```
    def increase_speed(self):
        """Increase speed settings."""
        self.ship_speed *= self.speedup_scale
        self.bullet_speed *= self.speedup_scale
        self.alien_speed *= 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:

alien_invasion.py

```
    def _check_bullet_alien_collisions(self):
        --snip--
        if not self.aliens:
            # Destroy existing bullets and create new fleet.
            self.bullets.empty()
            self._create_fleet()
            self.settings.increase_speed()
```

Changing the values of the speed settings `ship_speed`, `alien_speed`, and `bullet_speed` is enough to speed up the entire game!

Resetting the Speed

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

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
        button_clicked = self.play_button.rect.collidepoint(mouse_pos)
        if button_clicked and not self.stats.game_active:
            # Reset the game settings.
            self.settings.initialize_dynamic_settings()
            --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 285). Make the target move faster as the game progresses, and restart the target at the original speed when the player clicks Play.

14-4. Difficulty Levels: Make a set of buttons for *Alien Invasion* that allows the player to select an appropriate starting difficulty level for the game. Each button should assign the appropriate values for the attributes in `Settings` needed to create different difficulty levels.

Scoring

Let's implement a scoring system to track the game's score in real time and display the high score, level, and 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 eventually 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_game):
        """Initialize scorekeeping attributes."""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()
        self.settings = ai_game.settings
        self.stats = ai_game.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 `ai_game` parameter so it can access the `settings`, `screen`, and `stats` objects, which it will need to 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.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 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 and the text color to `render()`.

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 edge of the screen ❹. We then place the top edge 20 pixels down from the top of the screen ❺.

Then 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 onscreen at the location `score_rect` specifies.

Making a Scoreboard

To display the score, we'll create a `Scoreboard` instance in `AlienInvasion`. First, let's update the `import` statements:

alien_invasion.py

```
--snip--
from game_stats import GameStats
from scoreboard import Scoreboard
--snip--
```

Next, we make an instance of `Scoreboard` in `__init__()`:

alien_invasion.py

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

    # Create an instance to store game statistics,
    #   and create a scoreboard.
    self.stats = GameStats(self)
    self.sb = Scoreboard(self)
    --snip--
```

Then we draw the scoreboard onscreen in `_update_screen()`:

alien_invasion.py

```
def _update_screen(self):
    --snip--
    self.aliens.draw(self.screen)

    # Draw the score information.
    self.sb.show_score()

    # Draw the play button if the game is inactive.
    --snip--
```

We call `show_score()` just before we draw the Play button.

When you run *Alien Invasion* now, a 0 should appear at the top right of the screen. (At this point, we just want to make sure 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.

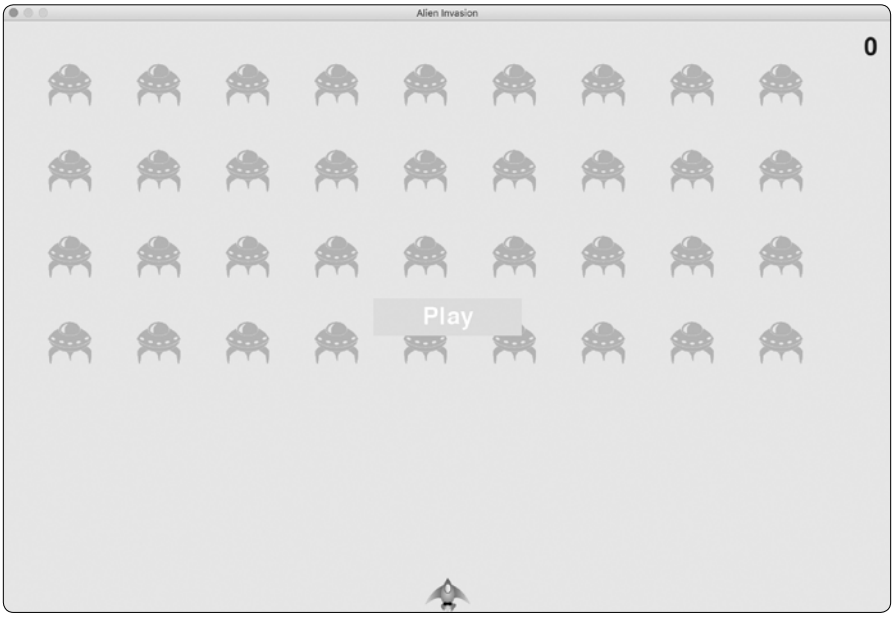


Figure 14-2: The score appears at the top-right corner of the screen.

Next, we'll assign point values to each alien!

Updating the Score as Aliens Are Shot Down

To write a live score onscreen, 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 each alien's point value 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()`.

Let's update the score each time an alien is shot down in `_check_bullet_alien_collisions()`:

alien_invasion.py

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

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

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. We then call `prep_score()` to create a new image for the updated score.

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

Resetting the Score

Right now, we're only prepping a new score *after* an alien has been hit, which works for most of the game. But we still see the old score when a new game starts until the first alien is hit in the new game.

We can fix this by prepping the score when starting a new game:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        --snip--
        if button_clicked and not self.stats.game_active:
            --snip--
            # Reset the game statistics.
            self.stats.reset_stats()
```

```
            self.stats.game_active = True
            self.sb.prep_score()
            --snip--
```

We call `prep_score()` after resetting the game stats when starting a new game. This preps the scoreboard with a 0 score.

Making Sure to Score All Hits

As currently written, our code could miss scoring for 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 only receive points for hitting one of the aliens. To fix this, let's refine the way that bullet and alien 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 values in the `collisions` dictionary to make sure we award points for each alien hit:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --snip--
    if collisions:
❶       for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
    --snip--
```

If the `collisions` dictionary has been defined, we loop through all values in the 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 its normal value.

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 initialize_dynamic_settings(self):
        --snip--

    def increase_speed(self):
        """Increase speed settings and alien point values."""
        self.ship_speed *= self.speedup_scale
        self.bullet_speed *= self.speedup_scale
        self.alien_speed *= 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 more challenging quickly. But to see a more notable difference in scoring, we need to change the alien point value by a larger amount (1.5). Now when we increase the game's speed, 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()` call to the `increase_speed()` method in `Settings`:

settings.py

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

The new point value should appear in the terminal every time you reach a new level.

NOTE *Be sure to remove the `print()` call after verifying that the point value is increasing, or it might affect your game's performance 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 scores. Also, let's 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 = round(self.stats.score, -1)
❷       score_str = "{:,}".format(rounded_score)
        self.score_image = self.font.render(score_str, True,
                self.text_color, self.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, when 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`.

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: A 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_game):
        --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()`.

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

scoreboard.py

```
    def __init__(self, ai_game):
        --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 = round(self.stats.high_score, -1)
        high_score_str = "{:,}".format(high_score)
❷       self.high_score_image = self.font.render(high_score_str, True,
                self.text_color, self.settings.bg_color)

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

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

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 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 method, `check_high_score()`, in `Scoreboard`:

scoreboard.py

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

The method `check_high_score()` checks 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 high score's image.

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

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --snip--
    if collisions:
        for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
        self.sb.check_high_score()
    --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 the current score and the 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.

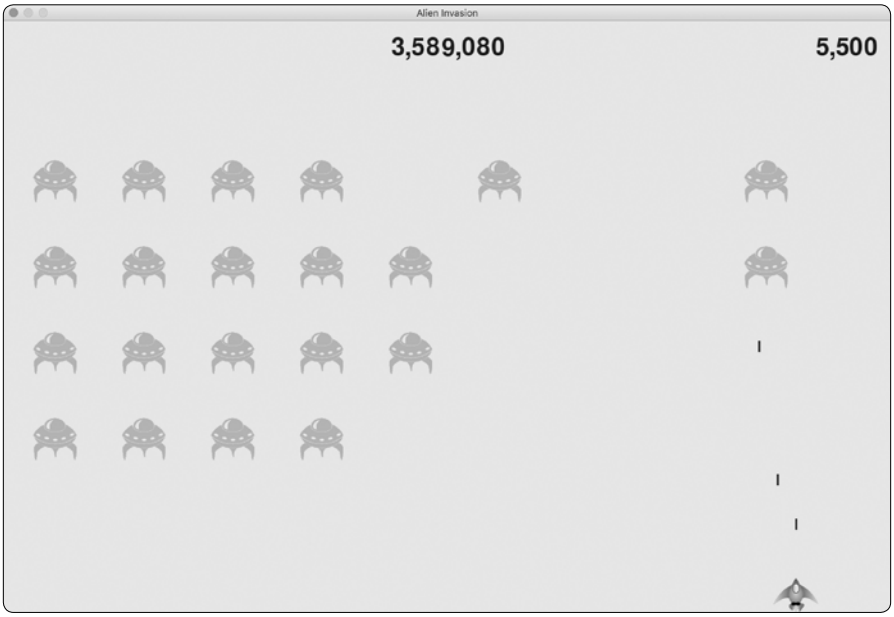


Figure 14-4: The high score is shown at the top center of the screen.

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.settings.ship_limit
```

```
        self.score = 0
        self.level = 1
```

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

scoreboard.py

```
    def __init__(self, ai_game):
        --snip--
        self.prep_high_score()
        self.prep_level()
```

Here's prep_level():

scoreboard.py

```
    def prep_level(self):
        """Turn the level into a rendered image."""
        level_str = str(self.stats.level)
❶       self.level_image = self.font.render(level_str, True,
                self.text_color, self.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 prep_level() method 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 level 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 new line draws the level image to the screen.

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

alien_invasion.py

```
    def _check_bullet_alien_collisions(self):
        --snip--
        if not self.aliens:
            # Destroy existing bullets and create new fleet.
            self.bullets.empty()
            self._create_fleet()
            self.settings.increase_speed()

            # Increase level.
            self.stats.level += 1
            self.sb.prep_level()
```

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

To ensure the level image updates properly at the start of a new game, we also call `prep_level()` when the player clicks the Play button:

alien_invasion.py

```
def _check_play_button(self, mouse_pos):
    --snip--
    if button_clicked and not self.stats.game_active:
        --snip--
        self.sb.prep_score()
        self.sb.prep_level()
        --snip--
```

We call `prep_level()` right after calling `prep_score()`.

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

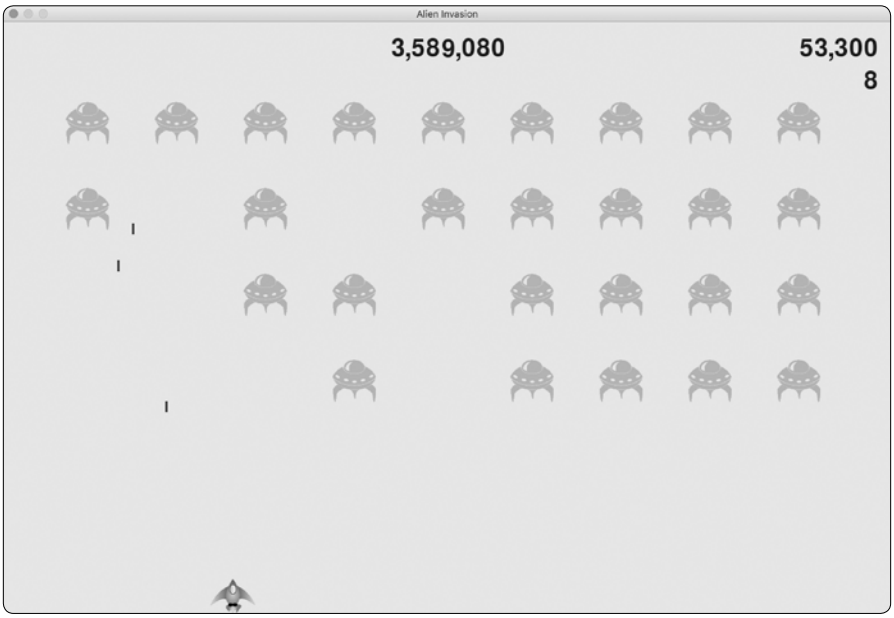


Figure 14-5: The current level appears just below the current score.

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, just as 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):
    """A class to manage the ship."""

    def __init__(self, ai_game):
        """Initialize the ship and set its starting position."""
❷       super().__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 are the `import` statements for `Scoreboard`:

scoreboard.py

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

from ship import Ship
```

Because we're making a group of ships, we import the `Group` and `Ship` classes.

Here's `__init__()`:

scoreboard.py

```
    def __init__(self, ai_game):
        """Initialize scorekeeping attributes."""
        self.ai_game = ai_game
        self.screen = ai_game.screen
        --snip--
        self.prep_level()
        self.prep_ships()
```

We assign the game instance to an attribute, because we'll need it to create some ships. 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_game)
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 appear in the upper-left corner of the screen ❹. Then 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):
        """Draw scores, level, 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)
        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 `AlienInvasion`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        --snip--
        if button_clicked and not self.stats.game_active:
            --snip--
            self.sb.prep_score()
            self.sb.prep_level()
            self.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:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to ship being hit by alien."""
        if self.stats.ships_left > 0:
            # Decrement ships_left, and update scoreboard.
            self.stats.ships_left -= 1
            self.sb.prep_ships()
            --snip--
```

We call `prep_ships()` after decreasing the value of `ships_left`, so the correct number of ships displays each time a ship is destroyed.

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

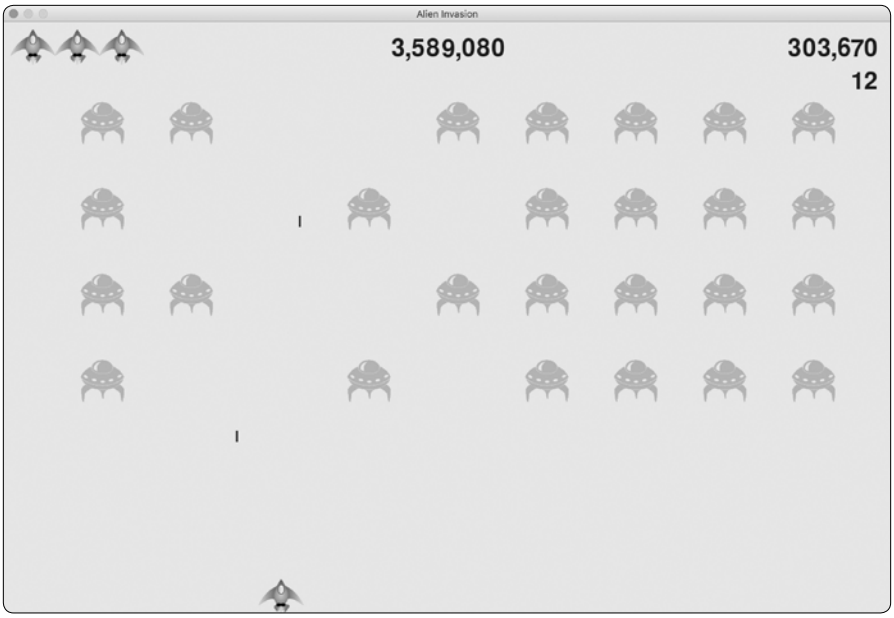


Figure 14-6: The complete scoring system for Alien Invasion

TRY IT YOURSELF

14-5. 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 in the high score when initializing its value in `GameStats`.

14-6. Refactoring: Look for methods that are doing more than one task, and refactor them to organize your code and make it 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 simplify `_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.*

(continued)

14-7. Expanding the Game: 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, such as explosions and shooting sounds.

14-8. Sideways Shooter, Final Version: Continue developing Sideways Shooter, using everything we've done in this project. Add a Play button, make the game speed up at appropriate points, and develop a scoring system. Be sure to refactor your code as you work, and look for opportunities to customize the game beyond what was shown in this chapter.

Summary

In this chapter, you learned how to implement a Play button to start a new game, 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, implement a progressive scoring system, and display information in textual and nontextual ways.

PROJECT 2

DATA VISUALIZATION

15

GENERATING DATA

Data visualization involves exploring data through visual representations. It's closely associated with *data analysis*, which uses code to explore the patterns and connections in a data set. A data set can be made up of a small list of numbers that fits in one line of code or it can be many gigabytes of data.

Making beautiful data representations is about more than pretty pictures. When a representation of a data set is simple and visually appealing, its meaning becomes clear to viewers. People will see patterns and significance in your data sets that they never knew existed.

Fortunately, you don't need a supercomputer to visualize complex data. With Python's efficiency, you can quickly explore data sets made of millions of individual data points on just a laptop. Also, the data points don't have to be numbers. With the basics you learned in the first part of this book, you can analyze nonnumerical data as well.

People use Python for data-intensive work in genetics, climate research, political and economic analysis, and much more. Data scientists have written

an impressive array of visualization and analysis tools in Python, many of which are available to you as well. One of the most popular tools is Matplotlib, a mathematical plotting library. We'll use Matplotlib to make simple plots, such as line graphs and scatter plots. Then we'll create a more interesting data set based on the concept of a random walk—a visualization generated from a series of random decisions.

We'll also use a package called Plotly, which creates visualizations that work well on digital devices. Plotly generates visualizations that automatically resize to fit a variety of display devices. These visualizations can also include a number of interactive features, such as emphasizing particular aspects of the data set when users hover over different parts of the visualization. We'll use Plotly to analyze the results of rolling dice.

Installing Matplotlib

To use Matplotlib for your initial set of visualizations, you'll need to install it using `pip`, a module that downloads and installs Python packages. Enter the following command at a terminal prompt:

```
$ python -m pip install --user matplotlib
```

This command tells Python to run the `pip` module and install the `matplotlib` package to the current user's Python installation. If you use a command other than `python` on your system to run programs or start a terminal session, such as `python3`, your command will look like this:

```
$ python3 -m pip install --user matplotlib
```

NOTE *If this command doesn't work on macOS, try running the command again without the `--user` flag.*

To see the kinds of visualizations you can make with Matplotlib, visit the sample gallery at *https://matplotlib.org/gallery/*. When you click a visualization in the gallery, you'll see the code used to generate the plot.

Plotting a Simple Line Graph

Let's plot a simple line graph using Matplotlib, and then customize it to create a more informative data visualization. We'll use the square number sequence 1, 4, 9, 16, 25 as the data for the graph.

Just provide Matplotlib with the numbers, as shown here, and Matplotlib should do the rest:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]
```

```
❶ fig, ax = plt.subplots()
  ax.plot(squares)

  plt.show()
```

We first import the `pyplot` module using the alias `plt` so we don't have to type `pyplot` repeatedly. (You'll see this convention often in online examples, so we'll do the same here.) The `pyplot` module contains a number of functions that generate charts and plots.

We create a list called `squares` to hold the data that we'll plot. Then we follow another common Matplotlib convention by calling the `subplots()` function ❶. This function can generate one or more plots in the same figure. The variable `fig` represents the entire figure or collection of plots that are generated. The variable `ax` represents a single plot in the figure and is the variable we'll use most of the time.

We then use the `plot()` method, which will try to plot the data it's given in a meaningful way. The function `plt.show()` opens Matplotlib's viewer and displays the plot, as shown in Figure 15-1. The viewer allows you to zoom and navigate the plot, and when you click the disk icon, you can save any plot images you like.

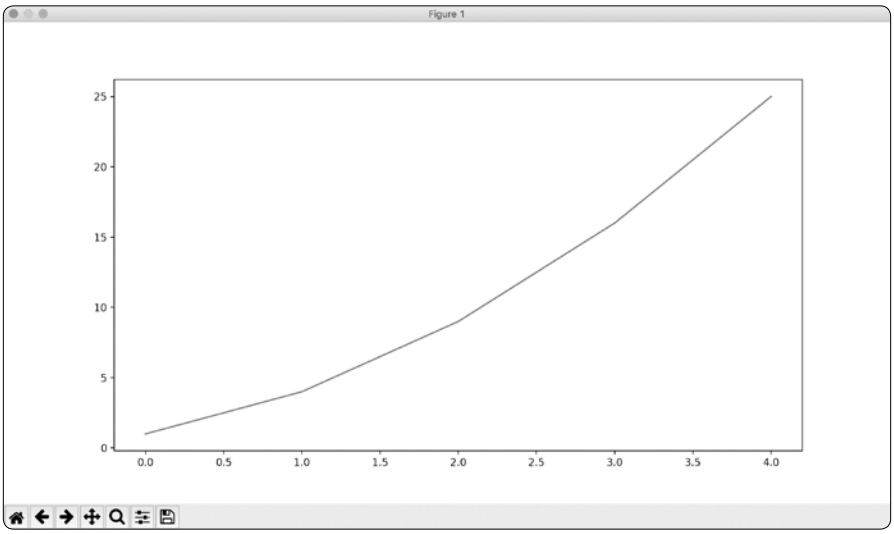


Figure 15-1: One of the simplest plots you can make in Matplotlib

Changing the Label Type and Line Thickness

Although the plot in Figure 15-1 shows that the numbers are increasing, the label type is too small and the line is a little thin to read easily. Fortunately, Matplotlib allows you to adjust every feature of a visualization.

We'll use a few of the available customizations to improve this plot's readability, as shown here:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
❶ ax.plot(squares, linewidth=3)

# Set chart title and label axes.
❷ ax.set_title("Square Numbers", fontsize=24)
❸ ax.set_xlabel("Value", fontsize=14)
ax.set_ylabel("Square of Value", fontsize=14)

# Set size of tick labels.
❹ ax.tick_params(axis='both', labelsize=14)

plt.show()
```

The `linewidth` parameter at ❶ controls the thickness of the line that `plot()` generates. The `set_title()` method at ❷ sets a title for the chart. The `fontsize` parameters, which appear repeatedly throughout the code, control the size of the text in various elements on the chart.

The `set_xlabel()` and `set_ylabel()` methods allow you to set a title for each of the axes ❸, and the method `tick_params()` styles the tick marks ❹. The arguments shown here affect the tick marks on both the x- and y-axes (`axis='both'`) and set the font size of the tick mark labels to 14 (`labelsize=14`).

As you can see in Figure 15-2, the resulting chart is much easier to read. The label type is bigger, and the line graph is thicker. It's often worth experimenting with these values to get an idea of what will look best in the resulting graph.

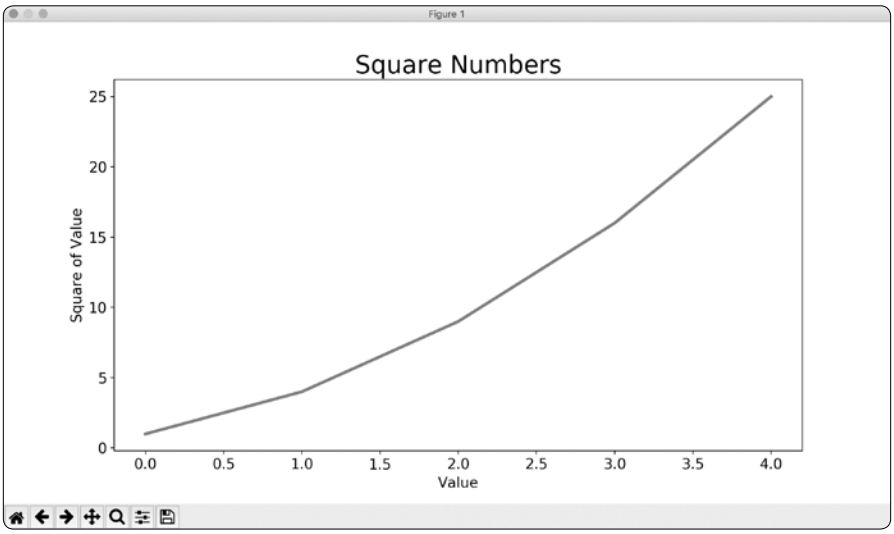


Figure 15-2: The chart is much easier to read now.

Correcting the Plot

But now that we can read the chart better, we see that the data is not plotted correctly. Notice at the end of the graph that the square of 4.0 is shown as 25! Let's fix that.

When you give `plot()` a sequence of numbers, it assumes the first data point corresponds to an x-coordinate value of 0, but our first point corresponds to an x-value of 1. We can override the default behavior by giving `plot()` the input and output values used to calculate the squares:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
ax.plot(input_values, squares, linewidth=3)

# Set chart title and label axes.
--snip--
```

Now `plot()` will graph the data correctly because we've provided the input and output values, so it doesn't have to assume how the output numbers were generated. The resulting plot, shown in Figure 15-3, is correct.

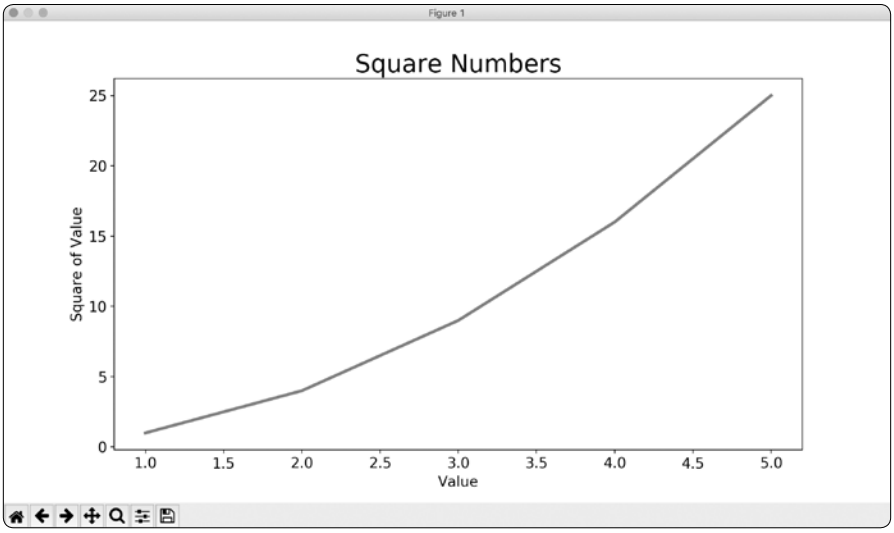


Figure 15-3: The data is now plotted correctly.

You can specify numerous arguments when using `plot()` and use a number of functions to customize your plots. We'll continue to explore these customization functions as we work with more interesting data sets throughout this chapter.

Using Built-in Styles

Matplotlib has a number of predefined styles available, with good starting settings for background colors, gridlines, line widths, fonts, font sizes, and more that will make your visualizations appealing without requiring much customization. To see the styles available on your system, run the following lines in a terminal session:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.available
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight',
--snip--
```

To use any of these styles, add one line of code before starting to generate the plot:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
--snip--
```

This code generates the plot shown in Figure 15-4. A wide variety of styles is available; play around with these styles to find some that you like.

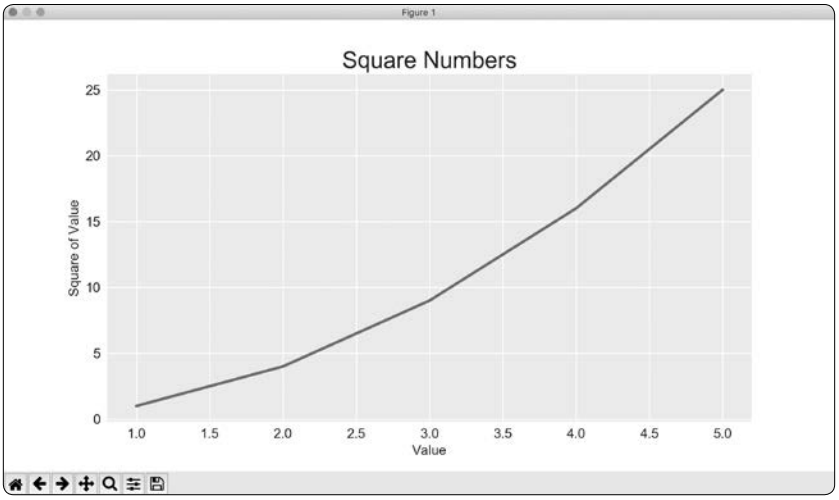


Figure 15-4: The built-in seaborn style

Plotting and Styling Individual Points with scatter()

Sometimes, it's useful to plot and style individual points based on certain characteristics. For example, you might plot small values in one color and larger values in a different color. You could also plot a large data set with

one set of styling options and then emphasize individual points by replotting them with different options.

To plot a single point, use the `scatter()` method. Pass the single (x, y) values of the point of interest to `scatter()` to plot those values:

scatter_squares.py

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(2, 4)

plt.show()
```

Let's style the output to make it more interesting. We'll add a title, label the axes, and make sure all the text is large enough to read:

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.scatter(2, 4, s=200)

# Set chart title and label axes.
ax.set_title("Square Numbers", fontsize=24)
ax.set_xlabel("Value", fontsize=14)
ax.set_ylabel("Square of Value", fontsize=14)

# Set size of tick labels.
ax.tick_params(axis='both', which='major', labelsize=14)

plt.show()
```

At ❶ we call `scatter()` and use the `s` argument to set the size of the dots used to draw the graph. When you run *scatter_squares.py* now, you should see a single point in the middle of the chart, as shown in Figure 15-5.

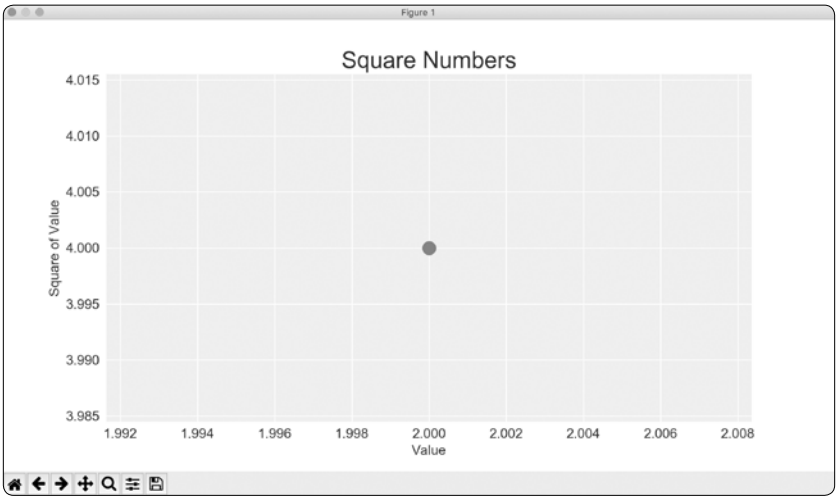


Figure 15-5: Plotting a single point

Plotting a Series of Points with scatter()

To plot a series of points, we can pass `scatter()` separate lists of x- and y-values, like this:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = [1, 2, 3, 4, 5]
y_values = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, s=100)

# Set chart title and label axes.
--snip--
```

The `x_values` list contains the numbers to be squared, and `y_values` contains the square of each number. When these lists are passed to `scatter()`, Matplotlib reads one value from each list as it plots each point. The points to be plotted are (1, 1), (2, 4), (3, 9), (4, 16), and (5, 25); Figure 15-6 shows the result.

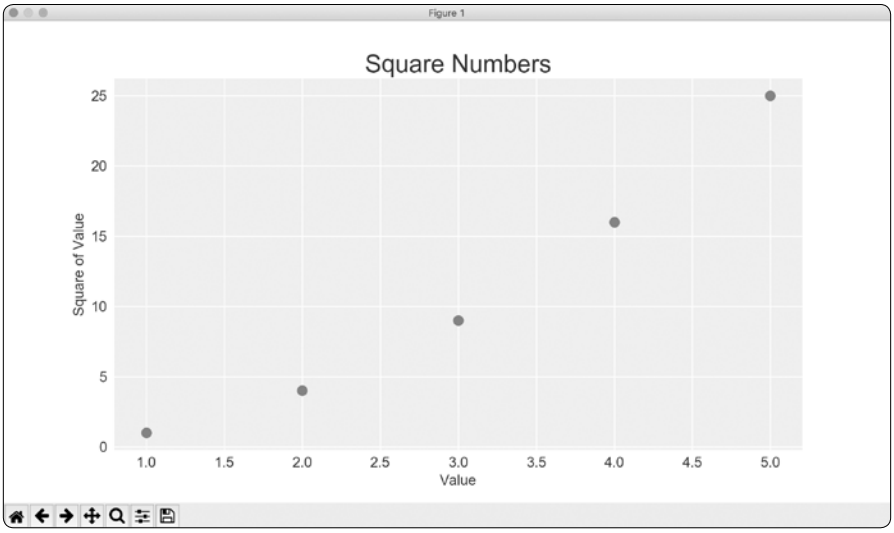


Figure 15-6: A scatter plot with multiple points

Calculating Data Automatically

Writing lists by hand can be inefficient, especially when we have many points. Rather than passing our points in a list, let's use a loop in Python to do the calculations for us.

Here's how this would look with 1000 points:

scatter_squares.py

```
import matplotlib.pyplot as plt

❶ x_values = range(1, 1001)
  y_values = [x**2 for x in x_values]

  plt.style.use('seaborn')
  fig, ax = plt.subplots()
❷ ax.scatter(x_values, y_values, s=10)

  # Set chart title and label axes.
  --snip--

  # Set the range for each axis.
❸ ax.axis([0, 1100, 0, 1100000])

  plt.show()
```

We start with a range of x-values containing the numbers 1 through 1000 ❶. Next, a list comprehension generates the y-values by looping through the x-values (`for x in x_values`), squaring each number (`x**2`) and storing the results in `y_values`. We then pass the input and output lists to `scatter()` ❷. Because this is a large data set, we use a smaller point size.

At ❸ we use the `axis()` method to specify the range of each axis. The `axis()` method requires four values: the minimum and maximum values for the x-axis and the y-axis. Here, we run the x-axis from 0 to 1100 and the y-axis from 0 to 1,100,000. Figure 15-7 shows the result.

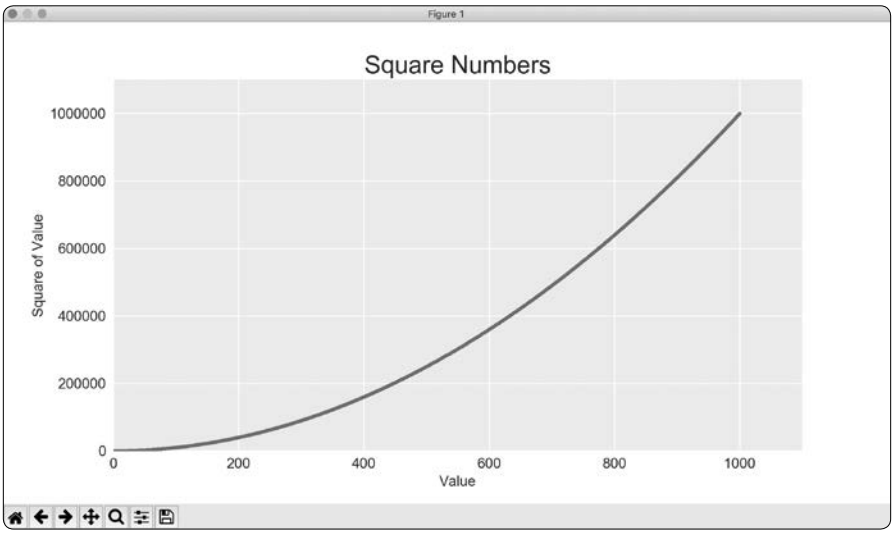


Figure 15-7: Python can plot 1000 points as easily as it plots 5 points.

Defining Custom Colors

To change the color of the points, pass `c` to `scatter()` with the name of a color to use in quotation marks, as shown here:

```
ax.scatter(x_values, y_values, c='red', s=10)
```

You can also define custom colors using the RGB color model. To define a color, pass the `c` argument a tuple with three decimal values (one each for red, green, and blue in that order), using values between 0 and 1. For example, the following line creates a plot with light-green dots:

```
ax.scatter(x_values, y_values, c=(0, 0.8, 0), s=10)
```

Values closer to 0 produce dark colors, and values closer to 1 produce lighter colors.

Using a Colormap

A *colormap* is a series of colors in a gradient that moves from a starting to an ending color. You use colormaps in visualizations to emphasize a pattern in the data. For example, you might make low values a light color and high values a darker color.

The `pyplot` module includes a set of built-in colormaps. To use one of these colormaps, you need to specify how `pyplot` should assign a color to each point in the data set. Here's how to assign each point a color based on its y-value:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = range(1, 1001)
y_values = [x**2 for x in x_values]

ax.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Blues, s=10)

# Set chart title and label axes.
--snip--
```

We pass the list of y-values to `c`, and then tell `pyplot` which colormap to use using the `cmap` argument. This code colors the points with lower y-values light blue and colors the points with higher y-values dark blue. Figure 15-8 shows the resulting plot.

NOTE *You can see all the colormaps available in `pyplot` at* https://matplotlib.org/*; go to* ***Examples****, scroll down to* ***Color****, and click* ***Colormap reference****.*

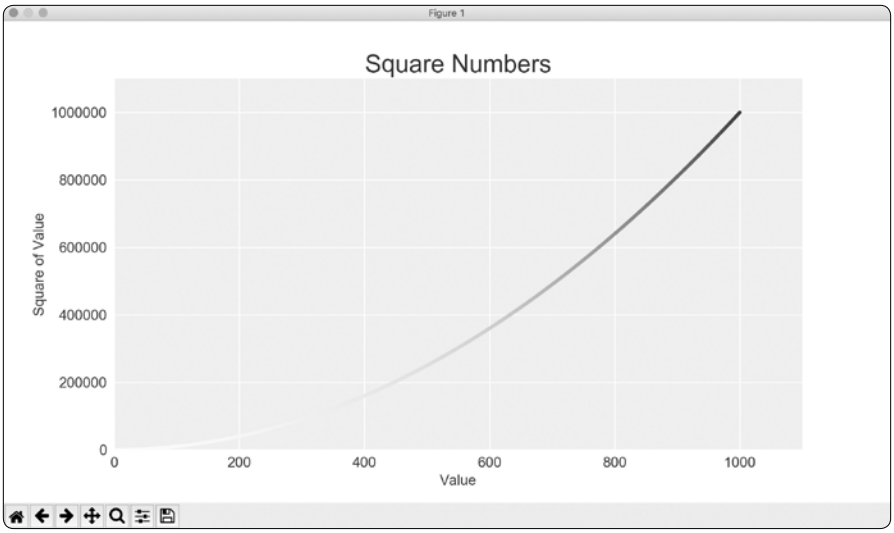


Figure 15-8: A plot using the `Blues` colormap

Saving Your Plots Automatically

If you want your program to automatically save the plot to a file, you can replace the call to `plt.show()` with a call to `plt.savefig()`:

```
plt.savefig('squares_plot.png', bbox_inches='tight')
```

The first argument is a filename for the plot image, which will be saved in the same directory as *scatter_squares.py*. The second argument trims extra whitespace from the plot. If you want the extra whitespace around the plot, just omit this argument.

TRY IT YOURSELF

15-1. Cubes: A number raised to the third power is a *cube*. Plot the first five cubic numbers, and then plot the first 5000 cubic numbers.

15-2. Colored Cubes: Apply a colormap to your cubes plot.

Random Walks

In this section, we'll use Python to generate data for a random walk, and then use Matplotlib to create a visually appealing representation of that data. A *random walk* is a path that has no clear direction but is determined by a series of random decisions, each of which is left entirely to chance. You might imagine a random walk as the path a confused ant would take if it took every step in a random direction.

Random walks have practical applications in nature, physics, biology, chemistry, and economics. For example, a pollen grain floating on a drop of water moves across the surface of the water because it's constantly pushed around by water molecules. Molecular motion in a water drop is random, so the path a pollen grain traces on the surface is a random walk. The code we'll write next models many real-world situations.

Creating the RandomWalk() Class

To create a random walk, we'll create a `RandomWalk` class, which will make random decisions about which direction the walk should take. The class needs three attributes: one variable to store the number of points in the walk and two lists to store the x- and y-coordinate values of each point in the walk.

We'll only need two methods for the `RandomWalk` class: the `__init__()` method and `fill_walk()`, which will calculate the points in the walk. Let's start with `__init__()` as shown here:

random_walk.py

```
❶ from random import choice

class RandomWalk:
    """A class to generate random walks."""

❷     def __init__(self, num_points=5000):
        """Initialize attributes of a walk."""
        self.num_points = num_points

        # All walks start at (0, 0).
❸       self.x_values = [0]
        self.y_values = [0]
```

To make random decisions, we'll store possible moves in a list and use the `choice()` function, from the `random` module, to decide which move to make each time a step is taken ❶. We then set the default number of points in a walk to 5000, which is large enough to generate some interesting patterns but small enough to generate walks quickly ❷. Then at ❸ we make two lists to hold the x- and y-values, and we start each walk at the point (0, 0).

Choosing Directions

We'll use the `fill_walk()` method, as shown here, to fill our walk with points and determine the direction of each step. Add this method to *random_walk.py*:

random_walk.py

```
    def fill_walk(self):
        """Calculate all the points in the walk."""

        # Keep taking steps until the walk reaches the desired length.
❶       while len(self.x_values) < self.num_points:

            # Decide which direction to go and how far to go in that direction.
❷           x_direction = choice([1, -1])
```

```
            x_distance = choice([0, 1, 2, 3, 4])
❸           x_step = x_direction * x_distance

            y_direction = choice([1, -1])
            y_distance = choice([0, 1, 2, 3, 4])
❹           y_step = y_direction * y_distance

            # Reject moves that go nowhere.
❺           if x_step == 0 and y_step == 0:
                continue

            # Calculate the new position.
❻           x = self.x_values[-1] + x_step
            y = self.y_values[-1] + y_step

            self.x_values.append(x)
            self.y_values.append(y)
```

At ❶ we set up a loop that runs until the walk is filled with the correct number of points. The main part of the `fill_walk()` method tells Python how to simulate four random decisions: will the walk go right or left? How far will it go in that direction? Will it go up or down? How far will it go in that direction?

We use `choice([1, -1])` to choose a value for `x_direction`, which returns either 1 for right movement or –1 for left ❷. Next, `choice([0, 1, 2, 3, 4])` tells Python how far to move in that direction (`x_distance`) by randomly selecting an integer between 0 and 4. (The inclusion of a 0 allows us to take steps along the y-axis as well as steps that have movement along both axes.)

At ❸ and ❹ we determine the length of each step in the *x* and *y* directions by multiplying the direction of movement by the distance chosen. A positive result for `x_step` means move right, a negative result means move left, and 0 means move vertically. A positive result for `y_step` means move up, negative means move down, and 0 means move horizontally. If the value of both `x_step` and `y_step` are 0, the walk doesn't go anywhere, so we continue the loop to ignore this move ❺.

To get the next x-value for the walk, we add the value in `x_step` to the last value stored in `x_values` ❻ and do the same for the y-values. When we have these values, we append them to `x_values` and `y_values`.

Plotting the Random Walk

Here's the code to plot all the points in the walk:

rw_visual.py

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Make a random walk.
❶ rw = RandomWalk()
rw.fill_walk()
```

```
# Plot the points in the walk.
plt.style.use('classic')
fig, ax = plt.subplots()
❷ ax.scatter(rw.x_values, rw.y_values, s=15)
plt.show()
```

We begin by importing `pyplot` and `RandomWalk`. We then create a random walk and store it in `rw` ❶, making sure to call `fill_walk()`. At ❷ we feed the walk's x- and y-values to `scatter()` and choose an appropriate dot size. Figure 15-9 shows the resulting plot with 5000 points. (The images in this section omit Matplotlib's viewer, but you'll continue to see it when you run *rw_visual.py*.)

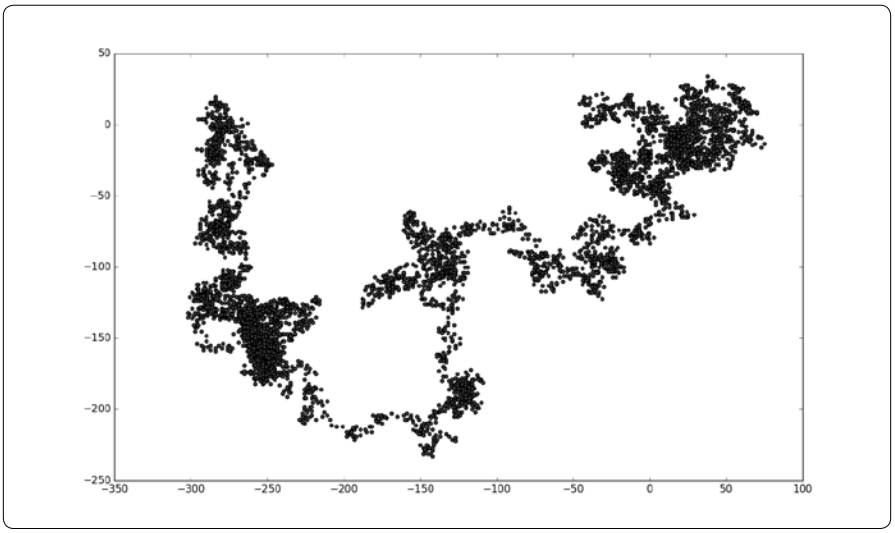


Figure 15-9: A random walk with 5000 points

Generating Multiple Random Walks

Every random walk is different, and it's fun to explore the various patterns that can be generated. One way to use the preceding code to make multiple walks without having to run the program several times is to wrap it in a `while` loop, like this:

rw_visual.py

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Keep making new walks, as long as the program is active.
while True:
    # Make a random walk.
    rw = RandomWalk()
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
```

```
    fig, ax = plt.subplots()
    ax.scatter(rw.x_values, rw.y_values, s=15)
    plt.show()

    keep_running = input("Make another walk? (y/n): ")
    if keep_running == 'n':
        break
```

This code generates a random walk, displays it in Matplotlib's viewer, and pauses with the viewer open. When you close the viewer, you'll be asked whether you want to generate another walk. Press **y** to generate walks that stay near the starting point, that wander off mostly in one direction, that have thin sections connecting larger groups of points, and so on. When you want to end the program, press **n**.

Styling the Walk

In this section, we'll customize our plots to emphasize the important characteristics of each walk and deemphasize distracting elements. To do so, we identify the characteristics we want to emphasize, such as where the walk began, where it ended, and the path taken. Next, we identify the characteristics to deemphasize, such as tick marks and labels. The result should be a simple visual representation that clearly communicates the path taken in each random walk.

Coloring the Points

We'll use a colormap to show the order of the points in the walk, and then remove the black outline from each dot so the color of the dots will be clearer. To color the points according to their position in the walk, we pass the `c` argument a list containing the position of each point. Because the points are plotted in order, the list just contains the numbers from 0 to 4999, as shown here:

rw_visual.py

```
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk()
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots()
❶   point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolors='none', s=15)
    plt.show()

    keep_running = input("Make another walk? (y/n): ")
    --snip--
```

At ❶ we use `range()` to generate a list of numbers equal to the number of points in the walk. Then we store them in the list `point_numbers`, which we'll use to set the color of each point in the walk. We pass `point_numbers` to the `c` argument, use the `Blues` colormap, and then pass `edgecolors='none'` to get rid of the black outline around each point. The result is a plot of the walk that varies from light to dark blue along a gradient, as shown in Figure 15-10.

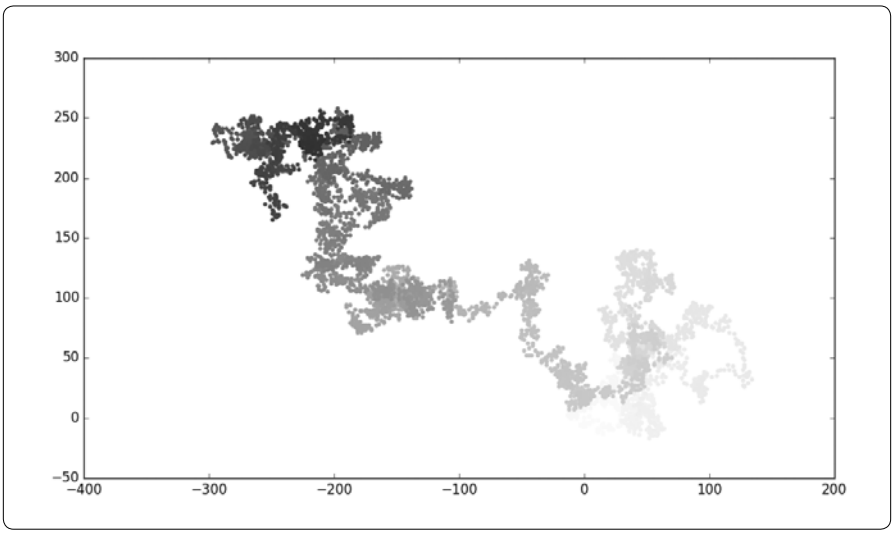


Figure 15-10: A random walk colored with the `Blues` colormap

Plotting the Starting and Ending Points

In addition to coloring points to show their position along the walk, it would be useful to see where each walk begins and ends. To do so, we can plot the first and last points individually after the main series has been plotted. We'll make the end points larger and color them differently to make them stand out, as shown here:

rw_visual.py

```
--snip--
while True:
    --snip--
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolors='none', s=15)

    # Emphasize the first and last points.
    ax.scatter(0, 0, c='green', edgecolors='none', s=100)
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
        s=100)

    plt.show()
    --snip--
```

To show the starting point, we plot point (0, 0) in green in a larger size (`s=100`) than the rest of the points. To mark the end point, we plot the last x- and y-value in the walk in red with a size of 100. Make sure you insert this code just before the call to `plt.show()` so the starting and ending points are drawn on top of all the other points.

When you run this code, you should be able to spot exactly where each walk begins and ends. (If these end points don't stand out clearly, adjust their color and size until they do.)

Cleaning Up the Axes

Let's remove the axes in this plot so they don't distract from the path of each walk. To turn off the axes, use this code:

rw_visual.py

```
--snip--
while True:
    --snip--
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
        s=100)

    # Remove the axes.
❶   ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    plt.show()
    --snip--
```

To modify the axes, we use the `ax.get_xaxis()` and `ax.get_yaxis()` methods ❶ to set the visibility of each axis to `False`. As you continue to work with visualizations, you'll frequently see this chaining of methods.

Run *rw_visual.py* now; you should see a series of plots with no axes.

Adding Plot Points

Let's increase the number of points to give us more data to work with. To do so, we increase the value of `num_points` when we make a `RandomWalk` instance and adjust the size of each dot when drawing the plot, as shown here:

rw_visual.py

```
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk(50_000)
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolor='none', s=1)
    --snip--
```

This example creates a random walk with 50,000 points (to mirror real-world data) and plots each point at size `s=1`. The resulting walk is wispy and cloud-like, as shown in Figure 15-11. As you can see, we've created a piece of art from a simple scatter plot!

Experiment with this code to see how much you can increase the number of points in a walk before your system starts to slow down significantly or the plot loses its visual appeal.

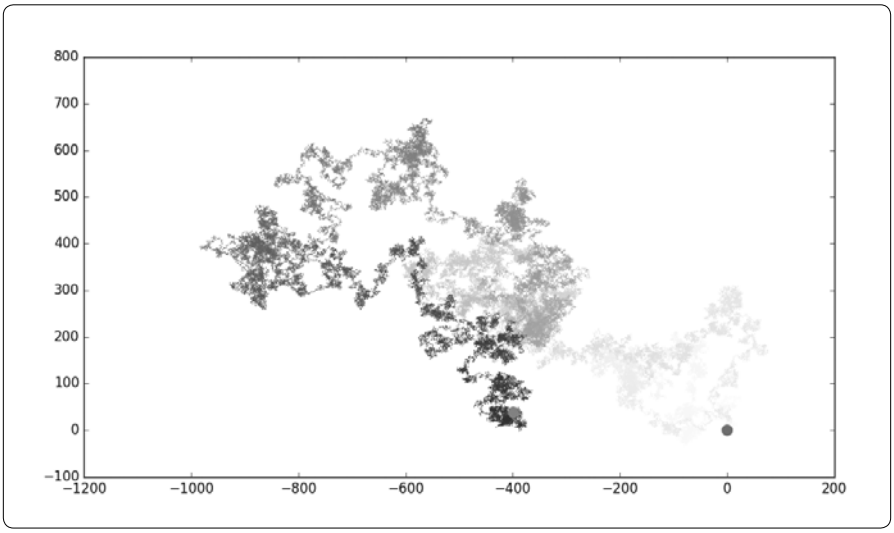


Figure 15-11: A walk with 50,000 points

Altering the Size to Fill the Screen

A visualization is much more effective at communicating patterns in data if it fits nicely on the screen. To make the plotting window better fit your screen, adjust the size of Matplotlib's output, like this:

rw_visual.py

```
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk(50_000)
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots(figsize=(15, 9))
    --snip--
```

When creating the plot, you can pass a `figsize` argument to set the size of the figure. The `figsize` parameter takes a tuple, which tells Matplotlib the dimensions of the plotting window in inches.

Matplotlib assumes that your screen resolution is 100 pixels per inch; if this code doesn't give you an accurate plot size, adjust the numbers as

necessary. Or, if you know your system's resolution, pass `plt.subplots()` the resolution using the `dpi` parameter to set a plot size that makes effective use of the space available on your screen, as shown here:

```
fig, ax = plt.subplots(figsize=(10, 6), dpi=128)
```

TRY IT YOURSELF

15-3. Molecular Motion: Modify *rw_visual.py* by replacing `plt.scatter()` with `plt.plot()`. To simulate the path of a pollen grain on the surface of a drop of water, pass in the `rw.x_values` and `rw.y_values`, and include a `linewidth` argument. Use 5000 instead of 50,000 points.

15-4. Modified Random Walks: In the `RandomWalk` class, `x_step` and `y_step` are generated from the same set of conditions. The direction is chosen randomly from the list `[1, -1]` and the distance from the list `[0, 1, 2, 3, 4]`. Modify the values in these lists to see what happens to the overall shape of your walks. Try a longer list of choices for the distance, such as 0 through 8, or remove the –1 from the *x* or *y* direction list.

15-5. Refactoring: The `fill_walk()` method is lengthy. Create a new method called `get_step()` to determine the direction and distance for each step, and then calculate the step. You should end up with two calls to `get_step()` in `fill_walk()`:

```
x_step = self.get_step()
y_step = self.get_step()
```

This refactoring should reduce the size of `fill_walk()` and make the method easier to read and understand.

Rolling Dice with Plotly

In this section, we'll use the Python package Plotly to produce interactive visualizations. Plotly is particularly useful when you're creating visualizations that will be displayed in a browser, because the visualizations will scale automatically to fit the viewer's screen. Visualizations that Plotly generates are also interactive; when the user hovers over certain elements on the screen, information about that element is highlighted.

In this project, we'll analyze the results of rolling dice. When you roll one regular, six-sided die, you have an equal chance of rolling any of the numbers from 1 through 6. However, when you use two dice, you're more likely to roll certain numbers rather than others. We'll try to determine

which numbers are most likely to occur by generating a data set that represents rolling dice. Then we'll plot the results of a large number of rolls to determine which results are more likely than others.

The study of rolling dice is often used in mathematics to explain various types of data analysis. But it also has real-world applications in casinos and other gambling scenarios, as well as in the way games like Monopoly and many role-playing games are played.

Installing Plotly

Install Plotly using `pip`, just as you did for Matplotlib:

```
$ python -m pip install --user plotly
```

If you used **`python3`** or something else when installing Matplotlib, make sure you use the same command here.

To see what kind of visualizations are possible with Plotly, visit the gallery of chart types at *https://plot.ly/python/*. Each example includes source code, so you can see how Plotly generates the visualizations.

Creating the Die Class

We'll create the following `Die` class to simulate the roll of one die:

die.py

```
from random import randint

class Die:
    """A class representing a single die."""

❶   def __init__(self, num_sides=6):
        """Assume a six-sided die."""
        self.num_sides = num_sides

    def roll(self):
        """Return a random value between 1 and number of sides."""
❷       return randint(1, self.num_sides)
```

The `__init__()` method takes one optional argument. With the `Die` class, when an instance of our die is created, the number of sides will always be six if no argument is included. If an argument *is* included, that value will set the number of sides on the die ❶. (Dice are named for their number of sides: a six-sided die is a D6, an eight-sided die is a D8, and so on.)

The `roll()` method uses the `randint()` function to return a random number between 1 and the number of sides ❷. This function can return the starting value (1), the ending value (`num_sides`), or any integer between the two.

Rolling the Die

Before creating a visualization based on the `Die` class, let's roll a D6, print the results, and check that the results look reasonable:

die_visual.py

```
from die import Die

# Create a D6.
❶ die = Die()

# Make some rolls, and store results in a list.
results = []
❷ for roll_num in range(100):
    result = die.roll()
    results.append(result)

print(results)
```

At ❶ we create an instance of `Die` with the default six sides. At ❷ we roll the die 100 times and store the results of each roll in the list `results`. Here's a sample set of results:

```
[4, 6, 5, 6, 1, 5, 6, 3, 5, 3, 5, 3, 2, 2, 1, 3, 1, 5, 3, 6, 3, 6, 5, 4,
 1, 1, 4, 2, 3, 6, 4, 2, 6, 4, 1, 3, 2, 5, 6, 3, 6, 2, 1, 1, 3, 4, 1, 4,
 3, 5, 1, 4, 5, 5, 2, 3, 3, 1, 2, 3, 5, 6, 2, 5, 6, 1, 3, 2, 1, 1, 1, 6,
 5, 5, 2, 2, 6, 4, 1, 4, 5, 1, 1, 1, 4, 5, 3, 3, 1, 3, 5, 4, 5, 6, 5, 4,
 1, 5, 1, 2]
```

A quick scan of these results shows that the `Die` class seems to be working. We see the values 1 and 6, so we know the smallest and largest possible values are being returned, and because we don't see 0 or 7, we know all the results are in the appropriate range. We also see each number from 1 through 6, which indicates that all possible outcomes are represented. Let's determine exactly how many times each number appears.

Analyzing the Results

We'll analyze the results of rolling one D6 by counting how many times we roll each number:

die_visual.py

```
--snip--
# Make some rolls, and store results in a list.
results = []
❶ for roll_num in range(1000):
    result = die.roll()
    results.append(result)

# Analyze the results.
frequencies = []
❷ for value in range(1, die.num_sides+1):
❸     frequency = results.count(value)
```

```
❹     frequencies.append(frequency)

print(frequencies)
```

Because we're no longer printing the results, we can increase the number of simulated rolls to 1000 ❶. To analyze the rolls, we create the empty list `frequencies` to store the number of times each value is rolled. We loop through the possible values (1 through 6 in this case) at ❷, count how many times each number appears in `results` ❸, and then append this value to the `frequencies` list ❹. We then print this list before making a visualization:

```
[155, 167, 168, 170, 159, 181]
```

These results look reasonable: we see six frequencies, one for each possible number when you roll a D6, and we see that no frequency is significantly higher than any other. Now let's visualize these results.

Making a Histogram

With a list of frequencies, we can make a *histogram* of the results. A histogram is a bar chart showing how often certain results occur. Here's the code to create the histogram:

die_visual.py

```
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die
--snip--

# Analyze the results.
frequencies = []
for value in range(1, die.num_sides+1):
    frequency = results.count(value)
    frequencies.append(frequency)

# Visualize the results.
❶ x_values = list(range(1, die.num_sides+1))
❷ data = [Bar(x=x_values, y=frequencies)]

❸ x_axis_config = {'title': 'Result'}
y_axis_config = {'title': 'Frequency of Result'}
❹ my_layout = Layout(title='Results of rolling one D6 1000 times',
        xaxis=x_axis_config, yaxis=y_axis_config)
❺ offline.plot({'data': data, 'layout': my_layout}, filename='d6.html')
```

To make a histogram, we need a bar for each of the possible results. We store these in a list called `x_values`, which starts at 1 and ends at the number of sides on the die ❶. Plotly doesn't accept the results of the `range()` function directly, so we need to convert the range to a list explicitly using

the `list()` function. The Plotly class `Bar()` represents a data set that will be formatted as a bar chart ❷. This class needs a list of x-values, and a list of y-values. The class must be wrapped in square brackets, because a data set can have multiple elements.

Each axis can be configured in a number of ways, and each configuration option is stored as an entry in a dictionary. At this point, we're just setting the title of each axis ❸. The `Layout()` class returns an object that specifies the layout and configuration of the graph as a whole ❹. Here we set the title of the graph and pass the x- and y-axis configuration dictionaries as well.

To generate the plot, we call the `offline.plot()` function ❺. This function needs a dictionary containing the data and layout objects, and it also accepts a name for the file where the graph will be saved. We store the output in a file called *d6.html*.

When you run the program *die_visual.py*, a browser will probably open showing the file *d6.html*. If this doesn't happen automatically, open a new tab in any web browser, and then open the file *d6.html* (in the folder where you saved *die_visual.py*). You should see a chart that looks like the one in Figure 15-12. (I've modified this chart slightly for printing; by default, Plotly generates charts with smaller text than what you see here.)

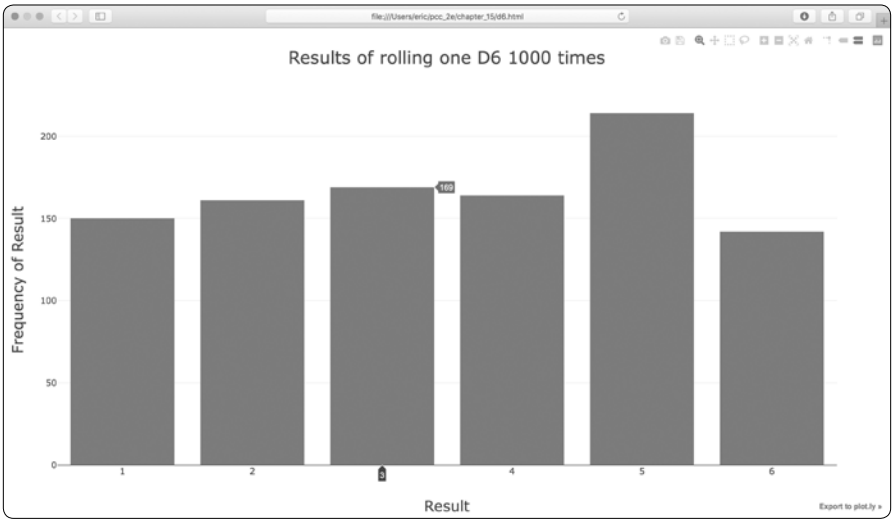


Figure 15-12: A simple bar chart created with Plotly

Notice that Plotly has made the chart interactive: hover your cursor over any bar in the chart, and you'll see the associated data. This feature is particularly useful when you're plotting multiple data sets on the same chart. Also notice the icons in the upper right, which allow you to pan and zoom the visualization, and save your visualization as an image.

Rolling Two Dice

Rolling two dice results in larger numbers and a different distribution of results. Let's modify our code to create two D6 dice to simulate the way we roll a pair of dice. Each time we roll the pair, we'll add the two numbers (one from each die) and store the sum in `results`. Save a copy of *die_visual.py* as *dice_visual.py*, and make the following changes:

dice_visual.py

```
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die

# Create two D6 dice.
die_1 = Die()
die_2 = Die()

# Make some rolls, and store results in a list.
results = []
for roll_num in range(1000):
❶     result = die_1.roll() + die_2.roll()
    results.append(result)

# Analyze the results.
frequencies = []
❷ max_result = die_1.num_sides + die_2.num_sides
❸ for value in range(2, max_result+1):
    frequency = results.count(value)
    frequencies.append(frequency)

# Visualize the results.
x_values = list(range(2, max_result+1))
data = [Bar(x=x_values, y=frequencies)]

❹ x_axis_config = {'title': 'Result', 'dtick': 1}
y_axis_config = {'title': 'Frequency of Result'}
my_layout = Layout(title='Results of rolling two D6 dice 1000 times',
        xaxis=x_axis_config, yaxis=y_axis_config)
offline.plot({'data': data, 'layout': my_layout}, filename='d6_d6.html')
```

After creating two instances of `Die`, we roll the dice and calculate the sum of the two dice for each roll ❶. The largest possible result (12) is the sum of the largest number on both dice, which we store in `max_result` ❷. The smallest possible result (2) is the sum of the smallest number on both dice. When we analyze the results, we count the number of results for each value between 2 and `max_result` ❸. (We could have used `range(2, 13)`, but this would work only for two D6 dice. When modeling real-world situations, it's best to write code that can easily model a variety of situations. This code allows us to simulate rolling a pair of dice with any number of sides.)

When creating the chart, we include the `dtick` key in the `x_axis_config` dictionary ❹. This setting controls the spacing between tick marks on the x-axis. Now that we have more bars on the histogram, Plotly's default

settings will only label some of the bars. The `'dtick': 1` setting tells Plotly to label every tick mark. We also update the title of the chart and change the output filename as well.

After running this code, you should see a chart that looks like the one in Figure 15-13.

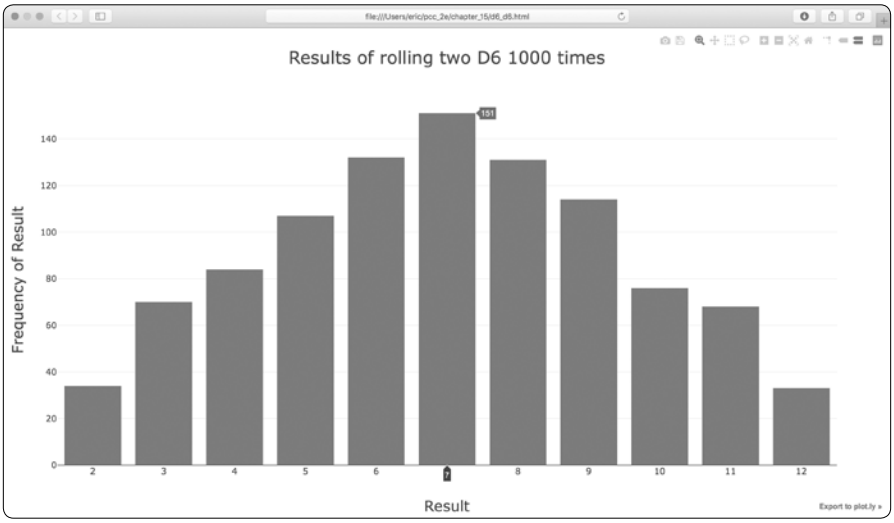


Figure 15-13: Simulated results of rolling two six-sided dice 1000 times

This graph shows the approximate results you're likely to get when you roll a pair of D6 dice. As you can see, you're least likely to roll a 2 or a 12 and most likely to roll a 7. This happens because there are six ways to roll a 7, namely: 1 and 6, 2 and 5, 3 and 4, 4 and 3, 5 and 2, or 6 and 1.

Rolling Dice of Different Sizes

Let's create a six-sided die and a ten-sided die, and see what happens when we roll them 50,000 times:

dice_visual.py

```
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die

# Create a D6 and a D10.
die_1 = Die()
❶ die_2 = Die(10)

# Make some rolls, and store results in a list.
results = []
for roll_num in range(50_000):
    result = die_1.roll() + die_2.roll()
    results.append(result)
```

```
# Analyze the results.
--snip--

# Visualize the results.
x_values = list(range(2, max_result+1))
data = [Bar(x=x_values, y=frequencies)]

x_axis_config = {'title': 'Result', 'dtick': 1}
y_axis_config = {'title': 'Frequency of Result'}
❷ my_layout = Layout(title='Results of rolling a D6 and a D10 50000 times',
        xaxis=x_axis_config, yaxis=y_axis_config)
offline.plot({'data': data, 'layout': my_layout}, filename='d6_d10.html')
```

To make a D10, we pass the argument `10` when creating the second `Die` instance ❶ and change the first loop to simulate 50,000 rolls instead of 1000. We change the title of the graph and update the output filename as well ❷.

Figure 15-14 shows the resulting chart. Instead of one most likely result, there are five. This happens because there's still only one way to roll the smallest value (1 and 1) and the largest value (6 and 10), but the smaller die limits the number of ways you can generate the middle numbers: there are six ways to roll a 7, 8, 9, 10, and 11. Therefore, these are the most common results, and you're equally likely to roll any one of these numbers.

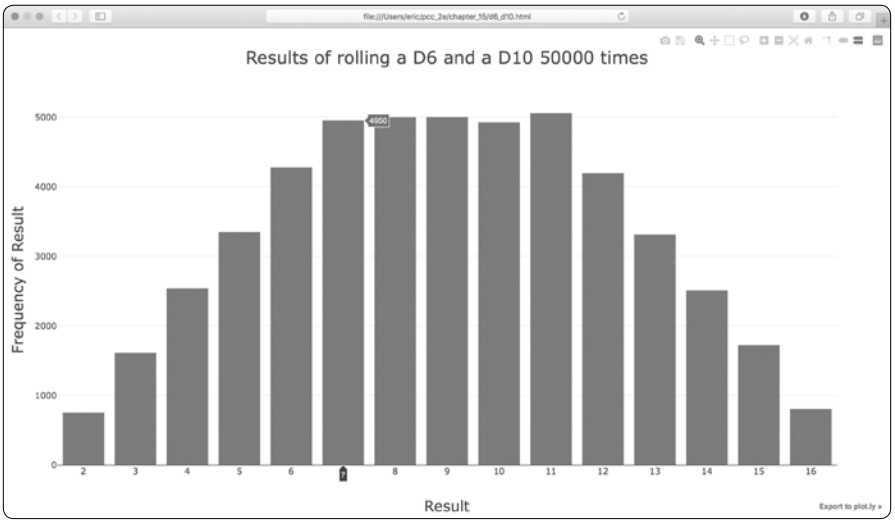


Figure 15-14: The results of rolling a six-sided die and a ten-sided die 50,000 times

Our ability to use Plotly to model the rolling of dice gives us considerable freedom in exploring this phenomenon. In just minutes you can simulate a tremendous number of rolls using a large variety of dice.

TRY IT YOURSELF

15-6. Two D8s: Create a simulation showing what happens when you roll two eight-sided dice 1000 times. Try to picture what you think the visualization will look like before you run the simulation; then see if your intuition was correct. Gradually increase the number of rolls until you start to see the limits of your system's capabilities.

15-7. Three Dice: When you roll three D6 dice, the smallest number you can roll is 3 and the largest number is 18. Create a visualization that shows what happens when you roll three D6 dice.

15-8. Multiplication: When you roll two dice, you usually add the two numbers together to get the result. Create a visualization that shows what happens if you multiply these numbers instead.

15-9. Die Comprehensions: For clarity, the listings in this section use the long form of `for` loops. If you're comfortable using list comprehensions, try writing a comprehension for one or both of the loops in each of these programs.

15-10. Practicing with Both Libraries: Try using Matplotlib to make a die-rolling visualization, and use Plotly to make the visualization for a random walk. (You'll need to consult the documentation for each library to complete this exercise.)

Summary

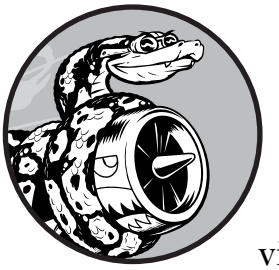
In this chapter, you learned to generate data sets and create visualizations of that data. You created simple plots with Matplotlib and used a scatter plot to explore random walks. You also created a histogram with Plotly and used a histogram to explore the results of rolling dice of different sizes.

Generating your own data sets with code is an interesting and powerful way to model and explore a wide variety of real-world situations. As you continue to work through the data visualization projects that follow, keep an eye out for situations you might be able to model with code. Look at the visualizations you see in news media, and see if you can identify those that were generated using methods similar to the ones you're learning in these projects.

In Chapter 16, you'll download data from online sources and continue to use Matplotlib and Plotly to explore that data.

16

DOWNLOADING DATA



In this chapter, you'll download data sets from online sources and create working visualizations of that data. You can find an incredible variety of data online, much of which hasn't been examined thoroughly. The ability to analyze this data allows you to discover patterns and connections that no one else has found.

We'll access and visualize data stored in two common data formats, CSV and JSON. We'll use Python's `csv` module to process weather data stored in the CSV (comma-separated values) format and analyze high and low temperatures over time in two different locations. We'll then use Matplotlib to generate a chart based on our downloaded data to display variations in temperature in two dissimilar environments: Sitka, Alaska, and Death Valley, California. Later in the chapter, we'll use the `json` module to access earthquake data stored in the JSON format and use Plotly to draw a world map showing the locations and magnitudes of recent earthquakes.

By the end of this chapter, you'll be prepared to work with different types and data set formats, and you'll have a deeper understanding of how to build complex visualizations. Being able to access and visualize online data of different types and formats is essential to working with a wide variety of real-world data sets.

The CSV File Format

One simple way to store data in a text file is to write the data as a series of values separated by commas, which is called *comma-separated values*. The resulting files are called *CSV* files. For example, here's a chunk of weather data in CSV format:

```
"USW00025333","SITKA AIRPORT, AK US","2018-01-01","0.45",,"48","38"
```

This is an excerpt of some weather data from January 1, 2018 in Sitka, Alaska. It includes the day's high and low temperatures, as well as a number of other measurements from that day. CSV files can be tricky for humans to read, but they're easy for programs to process and extract values from, which speeds up the data analysis process.

We'll begin with a small set of CSV-formatted weather data recorded in Sitka, which is available in the book's resources at *https://nostarch.com/pythoncrashcourse2e/*. Make a folder called *data* inside the folder where you're saving this chapter's programs. Copy the file *sitka_weather_07-2018_simple.csv* into this new folder. (After you download the book's resources, you'll have all the files you need for this project.)

NOTE *The weather data in this project was originally downloaded from* https://ncdc.noaa.gov/cdo-web/.

Parsing the CSV File Headers

Python's `csv` module in the standard library parses the lines in a CSV file and allows us to quickly extract the values we're interested in. Let's start by examining the first line of the file, which contains a series of headers for the data. These headers tell us what kind of information the data holds:

sitka_highs.py

```
import csv

filename = 'data/sitka_weather_07-2018_simple.csv'
❶ with open(filename) as f:
❷     reader = csv.reader(f)
❸     header_row = next(reader)
      print(header_row)
```

After importing the `csv` module, we assign the name of the file we're working with to `filename`. We then open the file and assign the resulting file

object to `f` ❶. Next, we call `csv.reader()` and pass it the file object as an argument to create a reader object associated with that file ❷. We assign the reader object to `reader`.

The `csv` module contains a `next()` function, which returns the next line in the file when passed the reader object. In the preceding listing, we call `next()` only once so we get the first line of the file, which contains the file headers ❸. We store the data that's returned in `header_row`. As you can see, `header_row` contains meaningful, weather-related headers that tell us what information each line of data holds:

```
['STATION', 'NAME', 'DATE', 'PRCP', 'TAVG', 'TMAX', 'TMIN']
```

The `reader` object processes the first line of comma-separated values in the file and stores each as an item in a list. The header `STATION` represents the code for the weather station that recorded this data. The position of this header tells us that the first value in each line will be the weather station code. The `NAME` header indicates that the second value in each line is the name of the weather station that made the recording. The rest of the headers specify what kinds of information were recorded in each reading. The data we're most interested in for now are the date, the high temperature (`TMAX`), and the low temperature (`TMIN`). This is a simple data set that contains only precipitation and temperature-related data. When you download your own weather data, you can choose to include a number of other measurements relating to wind speed, direction, and more detailed precipitation data.

Printing the Headers and Their Positions

To make it easier to understand the file header data, we print each header and its position in the list:

sitka_highs.py

```
--snip--
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

❶   for index, column_header in enumerate(header_row):
        print(index, column_header)
```

The `enumerate()` function returns both the index of each item and the value of each item as you loop through a list ❶. (Note that we've removed the line `print(header_row)` in favor of this more detailed version.)

Here's the output showing the index of each header:

```
0 STATION
1 NAME
2 DATE
3 PRCP
4 TAVG
5 TMAX
6 TMIN
```

Here we see that the dates and their high temperatures are stored in columns 2 and 5. To explore this data, we'll process each row of data in *sitka_weather_07-2018_simple.csv* and extract the values with the indexes 2 and 5.

Extracting and Reading Data

Now that we know which columns of data we need, let's read in some of that data. First, we'll read in the high temperature for each day:

sitka_highs.py

```
--snip--
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    # Get high temperatures from this file.
❶   highs = []
❷   for row in reader:
❸       high = int(row[5])
        highs.append(high)

print(highs)
```

We make an empty list called `highs` ❶ and then loop through the remaining rows in the file ❷. The `reader` object continues from where it left off in the CSV file and automatically returns each line following its current position. Because we've already read the header row, the loop will begin at the second line where the actual data begins. On each pass through the loop, we pull the data from index 5, which corresponds to the header `TMAX`, and assign it to the variable `high` ❸. We use the `int()` function to convert the data, which is stored as a string, to a numerical format so we can use it. We then append this value to `highs`.

The following listing shows the data now stored in `highs`:

```
[62, 58, 70, 70, 67, 59, 58, 62, 66, 59, 56, 63, 65, 58, 56, 59, 64, 60, 60,
 61, 65, 65, 63, 59, 64, 65, 68, 66, 64, 67, 65]
```

We've extracted the high temperature for each date and stored each value in a list. Now let's create a visualization of this data.

Plotting Data in a Temperature Chart

To visualize the temperature data we have, we'll first create a simple plot of the daily highs using Matplotlib, as shown here:

sitka_highs.py

```
import csv

import matplotlib.pyplot as plt

filename = 'data/sitka_weather_07-2018_simple.csv'
with open(filename) as f:
    --snip--
```

```
# Plot the high temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.plot(highs, c='red')

# Format plot.
❷ plt.title("Daily high temperatures, July 2018", fontsize=24)
❸ plt.xlabel('', fontsize=16)
plt.ylabel("Temperature (F)", fontsize=16)
plt.tick_params(axis='both', which='major', labelsize=16)

plt.show()
```

We pass the list of highs to `plot()` and pass `c='red'` to plot the points in red ❶. (We'll plot the highs in red and the lows in blue.) We then specify a few other formatting details, such as the title, font size, and labels ❷, which you should recognize from Chapter 15. Because we have yet to add the dates, we won't label the x-axis, but `plt.xlabel()` does modify the font size to make the default labels more readable ❸. Figure 16-1 shows the resulting plot: a simple line graph of the high temperatures for July 2018 in Sitka, Alaska.

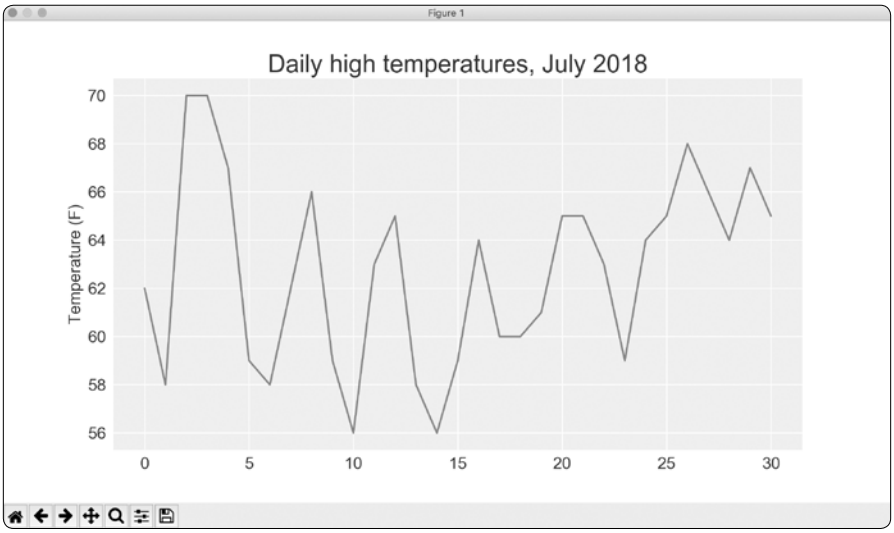


Figure 16-1: A line graph showing daily high temperatures for July 2018 in Sitka, Alaska

The datetime Module

Let's add dates to our graph to make it more useful. The first date from the weather data file is in the second row of the file:

```
"USW00025333","SITKA AIRPORT, AK US","2018-07-01","0.25",,"62","50"
```

The data will be read in as a string, so we need a way to convert the string "2018-07-01" to an object representing this date. We can construct an object representing July 1, 2018 using the strptime() method from the datetime module. Let's see how strptime() works in a terminal session:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2018-07-01', '%Y-%m-%d')
>>> print(first_date)
2018-07-01 00:00:00
```

We first import the datetime class from the datetime module. Then we call the method strptime() using the string containing the date we want to work with as its first argument. The second argument tells Python how the date is formatted. In this example, Python interprets '%Y-' to mean the part of the string before the first dash is a four-digit year; '%m-' means the part of the string before the second dash is a number representing the month; and '%d' means the last part of the string is the day of the month, from 1 to 31.

The strptime() method can take a variety of arguments to determine how to interpret the date. Table 16-1 shows some of these arguments.

Table 16-1: Date and Time Formatting Arguments from the datetime Module

Argument	Meaning
%A	Weekday name, such as *Monday*
%B	Month name, such as *January*
%m	Month, as a number (01 to 12)
%d	Day of the month, as a number (01 to 31)
%Y	Four-digit year, such as 2019
%y	Two-digit year, such as 19
%H	Hour, in 24-hour format (00 to 23)
%I	Hour, in 12-hour format (01 to 12)
%p	AM or PM
%M	Minutes (00 to 59)
%S	Seconds (00 to 61)

Plotting Dates

Now we can improve our temperature data plot by extracting dates for the daily highs and passing those highs and dates to plot(), as shown here:

sitka_highs.py

```
import csv
from datetime import datetime

import matplotlib.pyplot as plt

filename = 'data/sitka_weather_07-2018_simple.csv'
```

```
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    # Get dates and high temperatures from this file.
❶   dates, highs = [], []
    for row in reader:
❷       current_date = datetime.strptime(row[2], '%Y-%m-%d')
        high = int(row[5])
        dates.append(current_date)
        highs.append(high)

# Plot the high temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❸ ax.plot(dates, highs, c='red')

# Format plot.
plt.title("Daily high temperatures, July 2018", fontsize=24)
plt.xlabel('', fontsize=16)
❹ fig.autofmt_xdate()
plt.ylabel("Temperature (F)", fontsize=16)
plt.tick_params(axis='both', which='major', labelsize=16)

plt.show()
```

We create two empty lists to store the dates and high temperatures from the file ❶. We then convert the data containing the date information (`row[2]`) to a `datetime` object ❷ and append it to `dates`. We pass the dates and the high temperature values to `plot()` ❸. The call to `fig.autofmt_xdate()` ❹ draws the date labels diagonally to prevent them from overlapping. Figure 16-2 shows the improved graph.

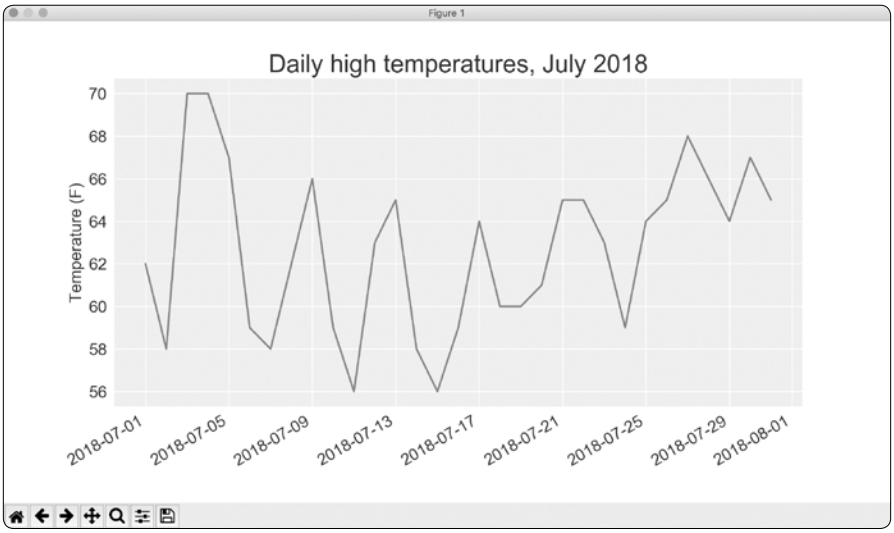


Figure 16-2: The graph is more meaningful now that it has dates on the x-axis.

Plotting a Longer Timeframe

With our graph set up, let's add more data to get a more complete picture of the weather in Sitka. Copy the file *sitka_weather_2018_simple.csv*, which contains a full year's worth of weather data for Sitka, to the folder where you're storing the data for this chapter's programs.

Now we can generate a graph for the entire year's weather:

sitka_highs.py

```
--snip--
❶ filename = 'data/sitka_weather_2018_simple.csv'
with open(filename) as f:
--snip--
# Format plot.
❷ plt.title("Daily high temperatures - 2018", fontsize=24)
plt.xlabel('', fontsize=16)
--snip--
```

We modify the filename to use the new data file *sitka_weather_2018_simple.csv* ❶, and we update the title of our plot to reflect the change in its content ❷. Figure 16-3 shows the resulting plot.

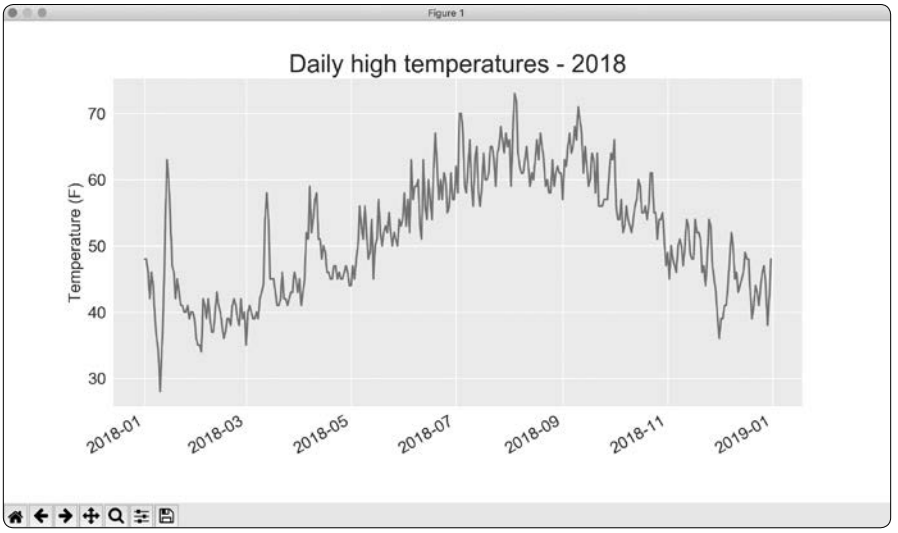


Figure 16-3: A year's worth of data

Plotting a Second Data Series

We can make our informative graph even more useful by including the low temperatures. We need to extract the low temperatures from the data file and then add them to our graph, as shown here:

sitka_highs_lows.py

```
--snip--
filename = 'sitka_weather_2018_simple.csv'
```

```
    with open(filename) as f:
        reader = csv.reader(f)
        header_row = next(reader)

        # Get dates, and high and low temperatures from this file.
❶       dates, highs, lows = [], [], []
        for row in reader:
            current_date = datetime.strptime(row[2], '%Y-%m-%d')
            high = int(row[5])
❷           low = int(row[6])
            dates.append(current_date)
            highs.append(high)
            lows.append(low)

    # Plot the high and low temperatures.
    plt.style.use('seaborn')
    fig, ax = plt.subplots()
    ax.plot(dates, highs, c='red')
❸   ax.plot(dates, lows, c='blue')

    # Format plot.
❹   plt.title("Daily high and low temperatures - 2018", fontsize=24)
    --snip--
```

At ❶ we add the empty list `lows` to hold low temperatures, and then extract and store the low temperature for each date from the seventh position in each row (`row[6]`) ❷. At ❸ we add a call to `plot()` for the low temperatures and color these values blue. Finally, we update the title ❹. Figure 16-4 shows the resulting chart.

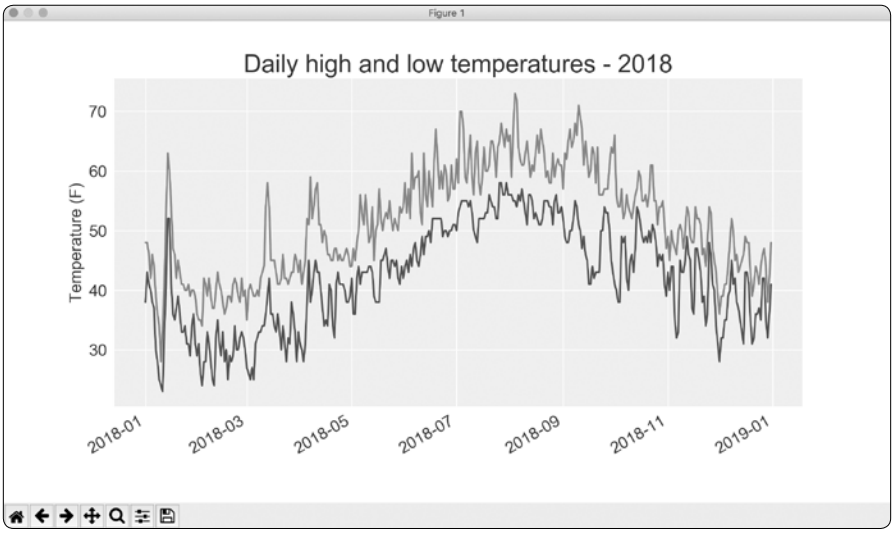


Figure 16-4: Two data series on the same plot

Shading an Area in the Chart

Having added two data series, we can now examine the range of temperatures for each day. Let's add a finishing touch to the graph by using shading to show the range between each day's high and low temperatures. To do so, we'll use the `fill_between()` method, which takes a series of x-values and two series of y-values, and fills the space between the two y-value series:

sitka_highs_lows.py

```
--snip--
# Plot the high and low temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.plot(dates, highs, c='red', alpha=0.5)
  ax.plot(dates, lows, c='blue', alpha=0.5)
❷ plt.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)
--snip--
```

The `alpha` argument at ❶ controls a color's transparency. An `alpha` value of 0 is completely transparent, and 1 (the default) is completely opaque. By setting `alpha` to 0.5, we make the red and blue plot lines appear lighter.

At ❷ we pass `fill_between()` the list `dates` for the x-values and then the two y-value series `highs` and `lows`. The `facecolor` argument determines the color of the shaded region; we give it a low `alpha` value of 0.1 so the filled region connects the two data series without distracting from the information they represent. Figure 16-5 shows the plot with the shaded region between the highs and lows.

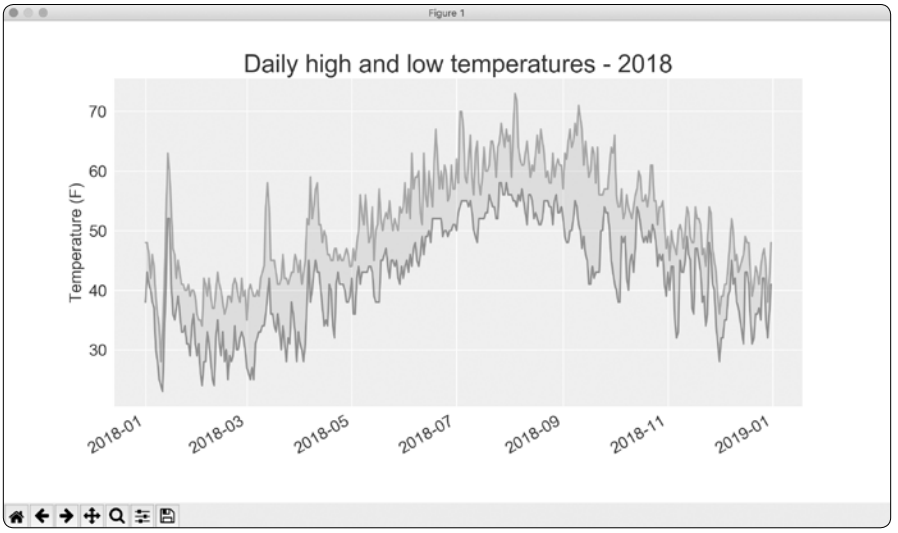


Figure 16-5: The region between the two data sets is shaded.

The shading helps make the range between the two data sets immediately apparent.

Error Checking

We should be able to run the *sitka_highs_lows.py* code using data for any location. But some weather stations collect different data than others, and some occasionally malfunction and fail to collect some of the data they're supposed to. Missing data can result in exceptions that crash our programs unless we handle them properly.

For example, let's see what happens when we attempt to generate a temperature plot for Death Valley, California. Copy the file *death_valley_2018_simple.csv* to the folder where you're storing the data for this chapter's programs.

First, let's run the code to see the headers that are included in this data file:

death_valley_highs_lows.py

```
import csv

filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    for index, column_header in enumerate(header_row):
        print(index, column_header)
```

Here's the output:

```
0 STATION
1 NAME
2 DATE
3 PRCP
4 TMAX
5 TMIN
6 TOBS
```

The date is in the same position at index 2. But the high and low temperatures are at indexes 4 and 5, so we'd need to change the indexes in our code to reflect these new positions. Instead of including an average temperature reading for the day, this station includes `TOBS`, a reading for a specific observation time.

I removed one of the temperature readings from this file to show what happens when some data is missing from a file. Change *sitka_highs_lows.py* to generate a graph for Death Valley using the indexes we just noted, and see what happens:

death_valley_highs_lows.py

```
--snip--
filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    --snip--
    # Get dates, and high and low temperatures from this file.
    dates, highs, lows = [], [], []
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
```

```
❶         high = int(row[4])
          low = int(row[5])
          dates.append(current_date)
--snip--
```

At ❶ we update the indexes to correspond to this file's `TMAX` and `TMIN` positions.

When we run the program, we get an error, as shown in the last line in the following output:

```
Traceback (most recent call last):
  File "death_valley_highs_lows.py", line 15, in <module>
    high = int(row[4])
ValueError: invalid literal for int() with base 10: ''
```

The traceback tells us that Python can't process the high temperature for one of the dates because it can't turn an empty string (`''`) into an integer. Rather than look through the data and finding out which reading is missing, we'll just handle cases of missing data directly.

We'll run error-checking code when the values are being read from the CSV file to handle exceptions that might arise. Here's how that works:

death_valley_highs_lows.py

```
--snip--
filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    --snip--
    for row in reader:
            current_date = datetime.strptime(row[2], '%Y-%m-%d')
❶           try:
                high = int(row[4])
                low = int(row[5])
            except ValueError:
❷               print(f"Missing data for {current_date}")
❸           else:
                dates.append(current_date)
                highs.append(high)
                lows.append(low)

# Plot the high and low temperatures.
--snip--

# Format plot.
❹ title = "Daily high and low temperatures - 2018\nDeath Valley, CA"
plt.title(title, fontsize=20)
plt.xlabel('', fontsize=16)
--snip--
```

Each time we examine a row, we try to extract the date and the high and low temperature ❶. If any data is missing, Python will raise a `ValueError` and we handle it by printing an error message that includes the date of the missing data ❷. After printing the error, the loop will continue processing the next row. If all data for a date is retrieved without error, the `else` block

will run and the data will be appended to the appropriate lists ❸. Because we're plotting information for a new location, we update the title to include the location on the plot, and we use a smaller font size to accommodate the longer title ❹.

When you run *death_valley_highs_lows.py* now, you'll see that only one date had missing data:

```
Missing data for 2018-02-18 00:00:00
```

Because the error is handled appropriately, our code is able to generate a plot, which skips over the missing data. Figure 16-6 shows the resulting plot.

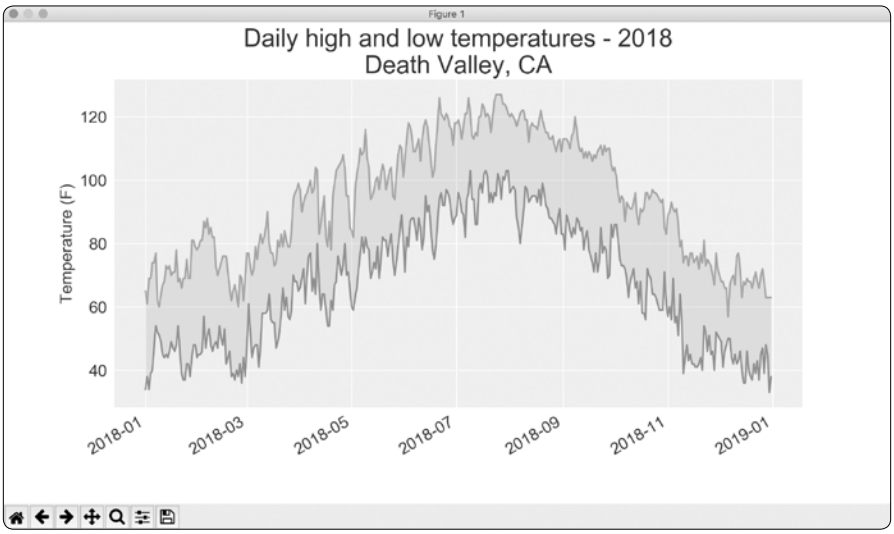


Figure 16-6: Daily high and low temperatures for Death Valley

Comparing this graph to the Sitka graph, we can see that Death Valley is warmer overall than southeast Alaska, as we expect. Also, the range of temperatures each day is greater in the desert. The height of the shaded region makes this clear.

Many data sets you work with will have missing, improperly formatted, or incorrect data. You can use the tools you learned in the first half of this book to handle these situations. Here we used a `try-except-else` block to handle missing data. Sometimes you'll use `continue` to skip over some data or use `remove()` or `del` to eliminate some data after it's been extracted. Use any approach that works, as long as the result is a meaningful, accurate visualization.

Downloading Your Own Data

If you want to download your own weather data, follow these steps:

1. Visit the NOAA Climate Data Online site at *https://www.ncdc.noaa.gov/cdo-web/*. In the *Discover Data By* section, click **Search Tool**. In the *Select a Dataset* box, choose **Daily Summaries**.

2. Select a date range, and in the *Search For* section, choose **ZIP Codes**. Enter the ZIP Code you're interested in, and click **Search**.
3. On the next page, you'll see a map and some information about the area you're focusing on. Below the location name, click **View Full Details**, or click the map and then click **Full Details**.
4. Scroll down and click **Station List** to see the weather stations that are available in this area. Choose one of the stations, and click **Add to Cart**. This data is free, even though the site uses a shopping cart icon. In the upper-right corner, click the cart.
5. In *Select the Output*, choose **Custom GHCN-Daily CSV**. Make sure the date range is correct, and click **Continue**.
6. On the next page, you can select the kinds of data you want. You can download one kind of data, for example, focusing on air temperature, or you can download all the data available from this station. Make your choices, and then click **Continue**.
7. On the last page, you'll see a summary of your order. Enter your email address, and click **Submit Order**. You'll receive a confirmation that your order was received, and in a few minutes you should receive another email with a link to download your data.

The data you download will be structured just like the data we worked with in this section. It might have different headers than those you saw in this section. But if you follow the same steps we used here, you should be able to generate visualizations of the data you're interested in.

TRY IT YOURSELF

16-1. Sitka Rainfall: Sitka is in a temperate rainforest, so it gets a fair amount of rainfall. In the data file *sitka_weather_2018_simple.csv* is a header called `PRCP`, which represents daily rainfall amounts. Make a visualization focusing on the data in this column. You can repeat the exercise for Death Valley if you're curious how little rainfall occurs in a desert.

16-2. Sitka–Death Valley Comparison: The temperature scales on the Sitka and Death Valley graphs reflect the different data ranges. To accurately compare the temperature range in Sitka to that of Death Valley, you need identical scales on the y-axis. Change the settings for the y-axis on one or both of the charts in Figures 16-5 and 16-6. Then make a direct comparison between temperature ranges in Sitka and Death Valley (or any two places you want to compare).

16-3. San Francisco: Are temperatures in San Francisco more like temperatures in Sitka or temperatures in Death Valley? Download some data for San Francisco, and generate a high-low temperature plot for San Francisco to make a comparison.

16-4. Automatic Indexes: In this section, we hardcoded the indexes corresponding to the `TMIN` and `TMAX` columns. Use the header row to determine the indexes for these values, so your program can work for Sitka or Death Valley. Use the station name to automatically generate an appropriate title for your graph as well.

16-5. Explore: Generate a few more visualizations that examine any other weather aspect you're interested in for any locations you're curious about.

Mapping Global Data Sets: JSON Format

In this section, you'll download a data set representing all the earthquakes that have occurred in the world during the previous month. Then you'll make a map showing the location of these earthquakes and how significant each one was. Because the data is stored in the JSON format, we'll work with it using the `json` module. Using Plotly's beginner-friendly mapping tool for location-based data, you'll create visualizations that clearly show the global distribution of earthquakes.

Downloading Earthquake Data

Copy the file *eq_1_day_m1.json* to the folder where you're storing the data for this chapter's programs. Earthquakes are categorized by their magnitude on the Richter scale. This file includes data for all earthquakes with a magnitude M1 or greater that took place in the last 24 hours (at the time of this writing). This data comes from one of the United States Geological Survey's earthquake data feeds, which you can find at *https://earthquake.usgs.gov/earthquakes/feed/*.

Examining JSON Data

When you open *eq_1_day_m1.json*, you'll see that it's very dense and hard to read:

```
{"type":"FeatureCollection","metadata":{"generated":1550361461000,...
{"type":"Feature","properties":{"mag":1.2,"place":"11km NNE of Nor...
{"type":"Feature","properties":{"mag":4.3,"place":"69km NNW of Ayn...
{"type":"Feature","properties":{"mag":3.6,"place":"126km SSE of Co...
{"type":"Feature","properties":{"mag":2.1,"place":"21km NNW of Teh...
{"type":"Feature","properties":{"mag":4,"place":"57km SSW of Kakto...
--snip--
```

This file is formatted more for machines than it is for humans. But we can see that the file contains some dictionaries, as well as information that we're interested in, such as earthquake magnitudes and locations.

The `json` module provides a variety of tools for exploring and working with JSON data. Some of these tools will help us reformat the file so we can look at the raw data more easily before we begin to work with it programmatically.

Let's start by loading the data and displaying it in a format that's easier to read. This is a long data file, so instead of printing it, we'll rewrite the data to a new file. Then we can open that file and scroll back and forth easily through the data:

eq_explore_data.py

```
import json

# Explore the structure of the data.
filename = 'data/eq_data_1_day_m1.json'
with open(filename) as f:
❶     all_eq_data = json.load(f)

❷ readable_file = 'data/readable_eq_data.json'
with open(readable_file, 'w') as f:
❸     json.dump(all_eq_data, f, indent=4)
```

We first import the `json` module to load the data properly from the file, and then store the entire set of data in `all_eq_data` ❶. The `json.load()` function converts the data into a format Python can work with: in this case, a giant dictionary. At ❷ we create a file to write this same data into a more readable format. The `json.dump()` function takes a JSON data object and a file object, and writes the data to that file ❸. The `indent=4` argument tells `dump()` to format the data using indentation that matches the data's structure.

When you look in your *data* directory and open the file *readable_eq_data.json*, here's the first part of what you'll see:

readable_eq_data.json

```
{
    "type": "FeatureCollection",
❶   "metadata": {
        "generated": 1550361461000,
        "url": "https://earthquake.usgs.gov/earthquakes/.../1.0_day.geojson",
        "title": "USGS Magnitude 1.0+ Earthquakes, Past Day",
        "status": 200,
        "api": "1.7.0",
        "count": 158
    },
❷   "features": [
    --snip--
```

The first part of the file includes a section with the key `"metadata"`. This tells us when the data file was generated and where we can find the data online. It also gives us a human-readable title and the number of earthquakes included in this file. In this 24-hour period, 158 earthquakes were recorded.

This *geoJSON* file has a structure that's helpful for location-based data. The information is stored in a list associated with the key `"features"` ❷. Because this file contains earthquake data, the data is in list form where every item in the list corresponds to a single earthquake. This structure might look confusing, but it's quite powerful. It allows geologists to store as much information as they need to in a dictionary about each earthquake, and then stuff all those dictionaries into one big list.

Let's look at a dictionary representing a single earthquake:

readable_eq_data.json

```
    --snip--
        {
            "type": "Feature",
❶           "properties": {
                "mag": 0.96,
                --snip--
❷               "title": "M 1.0 - 8km NE of Aguanga, CA"
            },
❸           "geometry": {
                "type": "Point",
                "coordinates": [
❹                   -116.7941667,
❺                   33.4863333,
                    3.22
                ]
            },
            "id": "ci37532978"
        },
```

The key `"properties"` contains a lot of information about each earthquake ❶. We're mainly interested in the magnitude of each quake, which is associated with the key `"mag"`. We're also interested in the title of each earthquake, which provides a nice summary of its magnitude and location ❷.

The key `"geometry"` helps us understand where the earthquake occurred ❸. We'll need this information to map each event. We can find the longitude ❹ and the latitude ❺ for each earthquake in a list associated with the key `"coordinates"`.

This file contains way more nesting than we'd use in the code we write, so if it looks confusing, don't worry: Python will handle most of the complexity. We'll only be working with one or two nesting levels at a time. We'll start by pulling out a dictionary for each earthquake that was recorded in the 24-hour time period.

NOTE *When we talk about locations, we often say the location's latitude first, followed by the longitude. This convention probably arose because humans discovered latitude long before we developed the concept of longitude. However, many geospatial frameworks list the longitude first and then the latitude, because this corresponds to the* (x, y) *convention we use in mathematical representations. The geoJSON format follows the (longitude, latitude) convention, and if you use a different framework it's important to learn what convention that framework follows.*

Making a List of All Earthquakes

First, we'll make a list that contains all the information about every earthquake that occurred.

eq_explore_data.py

```
import json

# Explore the structure of the data.
filename = 'data/eq_data_1_day_m1.json'
with open(filename) as f:
    all_eq_data = json.load(f)

all_eq_dicts = all_eq_data['features']
print(len(all_eq_dicts))
```

We take the data associated with the key `'features'` and store it in `all_eq_dicts`. We know this file contains records about 158 earthquakes, and the output verifies that we've captured all of the earthquakes in the file:

```
158
```

Notice how short this code is. The neatly formatted file *readable_eq_data.json* has over 6,000 lines. But in just a few lines, we can read through all that data and store it in a Python list. Next, we'll pull the magnitudes from each earthquake.

Extracting Magnitudes

Using the list containing data about each earthquake, we can loop through that list and extract any information we want. Now we'll pull the magnitude of each earthquake:

eq_explore_data.py

```
--snip--
all_eq_dicts = all_eq_data['features']

❶ mags = []
for eq_dict in all_eq_dicts:
❷     mag = eq_dict['properties']['mag']
    mags.append(mag)

print(mags[:10])
```

We make an empty list to store the magnitudes, and then loop through the dictionary `all_eq_dicts` ❶. Inside this loop, each earthquake is represented by the dictionary `eq_dict`. Each earthquake's magnitude is stored in the `'properties'` section of this dictionary under the key `'mag'` ❷. We store each magnitude in the variable `mag`, and then append it to the list `mags`.

We print the first 10 magnitudes, so we can see whether we're getting the correct data:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
```

Next, we'll pull the location data for each earthquake, and then we can make a map of the earthquakes.

Extracting Location Data

The location data is stored under the key `"geometry"`. Inside the geometry dictionary is a `"coordinates"` key, and the first two values in this list are the longitude and latitude. Here's how we'll pull this data:

eq_explore_data.py

```
--snip--
all_eq_dicts = all_eq_data['features']

mags, lons, lats = [], [], []
for eq_dict in all_eq_dicts:
    mag = eq_dict['properties']['mag']
❶   lon = eq_dict['geometry']['coordinates'][0]
    lat = eq_dict['geometry']['coordinates'][1]
    mags.append(mag)
    lons.append(lon)
    lats.append(lat)

print(mags[:10])
print(lons[:5])
print(lats[:5])
```

We make empty lists for the longitudes and latitudes. The code `eq_dict['geometry']` accesses the dictionary representing the geometry element of the earthquake ❶. The second key, `'coordinates'`, pulls the list of values associated with `'coordinates'`. Finally, the 0 index asks for the first value in the list of coordinates, which corresponds to an earthquake's longitude.

When we print the first five longitudes and latitudes, the output shows that we're pulling the correct data:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
[-116.7941667, -148.9865, -74.2343, -161.6801, -118.5316667]
[33.4863333, 64.6673, -12.1025, 54.2232, 35.3098333]
```

With this data, we can move on to mapping each earthquake.

Building a World Map

With the information we've pulled so far, we can build a simple world map. Although it won't look presentable yet, we want to make sure the information is displayed correctly before focusing on style and presentation issues. Here's the initial map:

eq_world_map.py

```
import json

❶ from plotly.graph_objs import Scattergeo, Layout
from plotly import offline

--snip--
```

```
    for eq_dict in all_eq_dicts:
        --snip--

    # Map the earthquakes.
❷ data = [Scattergeo(lon=lons, lat=lats)]
❸ my_layout = Layout(title='Global Earthquakes')

❹ fig = {'data': data, 'layout': my_layout}
    offline.plot(fig, filename='global_earthquakes.html')
```

We import the `Scattergeo` chart type and the `Layout` class, and then import the `offline` module to render the map ❶. As we did when making a bar chart, we define a list called `data`. We create the `Scattergeo` object inside this list ❷, because you can plot more than one data set on any visualization you make. A `Scattergeo` chart type allows you to overlay a scatter plot of geographic data on a map. In the simplest use of this chart type, you only need to provide a list of longitudes and a list of latitudes.

We give the chart an appropriate title ❸ and create a dictionary called `fig` that contains the data and the layout ❹. Finally, we pass `fig` to the `plot()` function along with a descriptive filename for the output. When you run this file, you should see a map that looks like the one in Figure 16-7. Earthquakes usually occur near plate boundaries, which matches what we see in the chart.

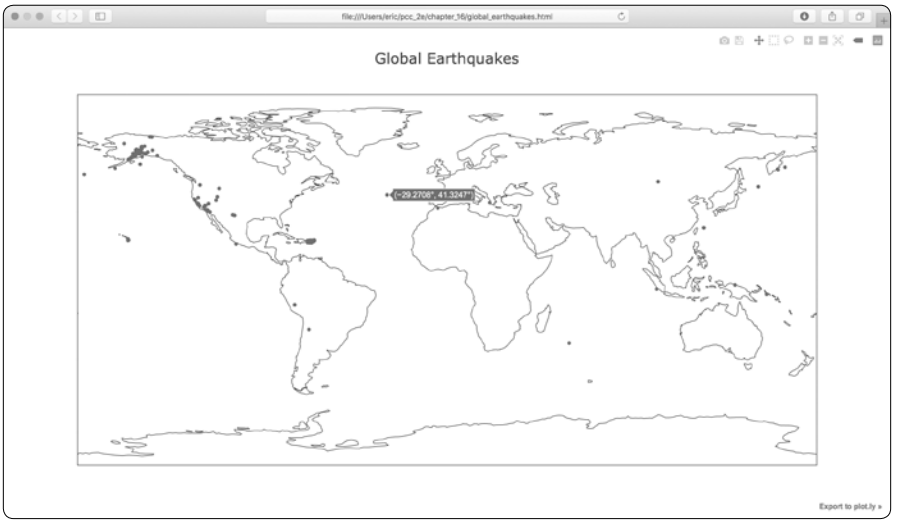


Figure 16-7: A simple map showing where all the earthquakes in the last 24 hours occurred

We can do a lot of modifications to make this map more meaningful and easier to read, so let's make some of these changes.

A Different Way of Specifying Chart Data

Before we configure the chart, let's look at a slightly different way to specify the data for a Plotly chart. In the current chart, the `data` list is defined in one line:

```
data = [Scattergeo(lon=lons, lat=lats)]
```

This is one of the simplest ways to define the data for a chart in Plotly. But it's not the best way when you want to customize the presentation. Here's an equivalent way to define the data for the current chart:

```
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
}]
```

In this approach, all the information about the data is structured as key-value pairs in a dictionary. If you put this code into *eq_plot.py*, you'll see the same chart we just generated. This format allows us to specify customizations more easily than the previous format.

Customizing Marker Size

When we're figuring out how to improve the map's styling, we should focus on aspects of the data that we want to communicate more clearly. The current map shows the location of each earthquake, but it doesn't communicate the severity of any earthquake. We want viewers to immediately see where the most significant earthquakes occur in the world.

To do this, we'll change the size of markers depending on the magnitude of each earthquake:

eq_world_map.py

```
import json
--snip--
# Map the earthquakes.
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
❶   'marker': {
❷       'size': [5*mag for mag in mags],
    },
}]
my_layout = Layout(title='Global Earthquakes')
--snip--
```

Plotly offers a huge variety of customizations you can make to a data series, each of which can be expressed as a key-value pair. Here we're using the key `'marker'` to specify how big each marker on the map should be ❶. We use a nested dictionary as the value associated with `'marker'`, because you can specify a number of settings for all the markers in a series.

We want the size to correspond to the magnitude of each earthquake. But if we just pass in the `mags` list, the markers would be too small to easily see the size differences. We need to multiply the magnitude by a scale factor to get an appropriate marker size. On my screen, a value of 5 works well; a slightly smaller or larger value might work better for your map. We use a list comprehension, which generates an appropriate marker size for each value in the `mags` list ❷.

When you run this code, you should see a map that looks like the one in Figure 16-8. This is a much better map, but we can still do more.

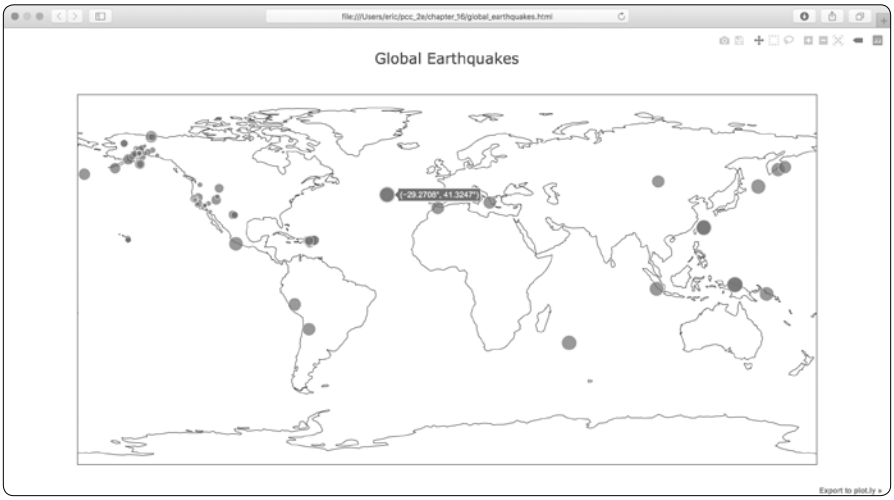


Figure 16-8: The map now shows the magnitude of each earthquake.

Customizing Marker Colors

We can also customize each marker's color to provide some classification to the severity of each earthquake. We'll use Plotly's colorscales to do this. Before you make these changes, copy the file *eq_data_30_day_m1.json* to your data directory. This file includes earthquake data for a 30-day period, and the map will be much more interesting to look at using this larger data set.

Here's how to use a colorscale to represent the magnitude of each earthquake:

eq_world_map.py

```
--snip--
❶ filename = 'data/eq_data_30_day_m1.json'
--snip--
# Map the earthquakes.
data = [{
    --snip--
```

```
        'marker': {
            'size': [5*mag for mag in mags],
❷           'color': mags,
❸           'colorscale': 'Viridis',
❹           'reversescale': True,
❺           'colorbar': {'title': 'Magnitude'},
        },
    }]
    --snip--
```

Be sure to update the filename so you're using the 30-day data set ❶. All the significant changes here occur in the `'marker'` dictionary, because we're only modifying the markers' appearance. The `'color'` setting tells Plotly what values it should use to determine where each marker falls on the colorscale ❷. We use the `mags` list to determine the color that's used. The `'colorscale'` setting tells Plotly which range of colors to use: `'Viridis'` is a colorscale that ranges from dark blue to bright yellow and works well for this data set ❸. We set `'reversescale'` to `True`, because we want to use bright yellow for the lowest values and dark blue for the most severe earthquakes ❹. The `'colorbar'` setting allows us to control the appearance of the colorscale shown on the side of the map. Here we title the colorscale `'Magnitude'` to make it clear what the colors represent ❺.

When you run the program now, you'll see a much nicer-looking map. In Figure 16-9, the colorscale shows the severity of individual earthquakes. Plotting this many earthquakes really makes it clear where the tectonic plate boundaries are!

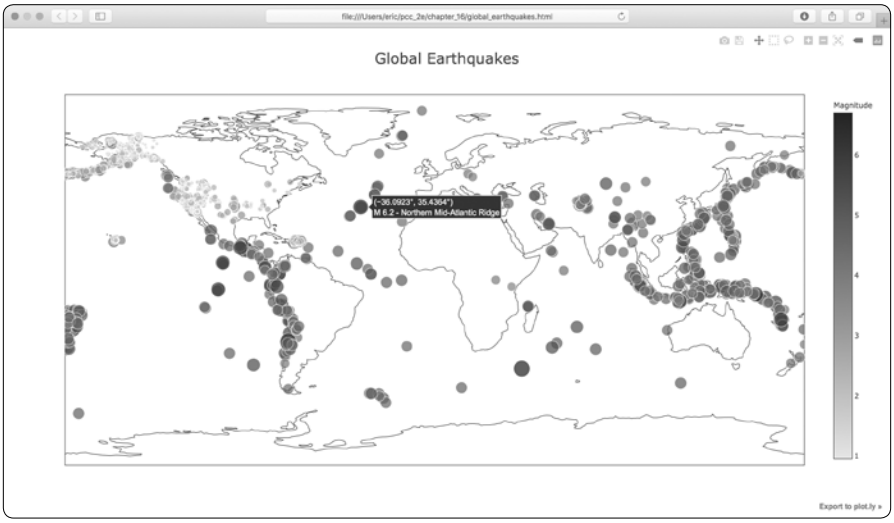


Figure 16-9: In 30 days' worth of earthquakes, color and size are used to represent the magnitude of each earthquake.

Other Colorscales

You can also choose from a number of other colorscales. To see the available colorscales, save the following short program as *show_color_scales.py*:

show_color_scales.py

```
from plotly import colors

for key in colors.PLOTLY_SCALES.keys():
    print(key)
```

Plotly stores the colorscales in the `colors` module. The colorscales are defined in the dictionary `PLOTLY_SCALES`, and the names of the colorscales serve as the keys in the dictionary. Here's the output showing all the available colorscales:

```
Greys
YlGnBu
Greens
--snip--
Viridis
```

Feel free to try out these colorscales; remember that you can reverse any of these scales using the `reversescale` setting.

NOTE *If you print the `PLOTLY_SCALES` dictionary, you can see how colorscales are defined. Every scale has a beginning color and an end color, and some scales have one or more intermediate colors defined as well. Plotly interpolates shades between each of these defined colors.*

Adding Hover Text

To finish this map, we'll add some informative text that appears when you hover over the marker representing an earthquake. In addition to showing the longitude and latitude, which appear by default, we'll show the magnitude and provide a description of the approximate location as well.

To make this change, we need to pull a little more data from the file and add it to the dictionary in `data` as well:

eq_world_map.py

```
--snip--
❶ mags, lons, lats, hover_texts = [], [], [], []
for eq_dict in all_eq_dicts:
    --snip--
    lat = eq_dict['geometry']['coordinates'][1]
❷   title = eq_dict['properties']['title']
    mags.append(mag)
    lons.append(lon)
    lats.append(lat)
    hover_texts.append(title)
--snip--
```

```
# Map the earthquakes.
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
❸   'text': hover_texts,
    'marker': {
        --snip--
    },
}]
--snip--
```

We first make a list called `hover_texts` to store the label we'll use for each marker ❶. The "title" section of the earthquake data contains a descriptive name of the magnitude and location of each earthquake in addition to its longitude and latitude. At ❷ we pull this information and assign it to the variable `title`, and then append it to the list `hover_texts`.

When we include the key `'text'` in the `data` object, Plotly uses this value as a label for each marker when the viewer hovers over the marker. When we pass a list that matches the number of markers, Plotly pulls an individual label for each marker it generates ❸. When you run this program, you should be able to hover over any marker, see a description of where that earthquake took place, and read its exact magnitude.

This is impressive! In approximately 40 lines of code, we've created a visually appealing and meaningful map of global earthquake activity that also illustrates the geological structure of the planet. Plotly offers a wide range of ways you can customize the appearance and behavior of your visualizations. Using Plotly's many options, you can make charts and maps that show exactly what you want them to.

TRY IT YOURSELF

16-6. Refactoring: The loop that pulls data from `all_eq_dicts` uses variables for the magnitude, longitude, latitude, and title of each earthquake before appending these values to their appropriate lists. This approach was chosen for clarity in how to pull data from a JSON file, but it's not necessary in your code. Instead of using these temporary variables, pull each value from `eq_dict` and append it to the appropriate list in one line. Doing so should shorten the body of this loop to just four lines.

16-7. Automated Title: In this section, we specified the title manually when defining `my_layout`, which means we have to remember to update the title every time the source file changes. Instead, you can use the title for the data set in the metadata part of the JSON file. Pull this value, assign it to a variable, and use this for the title of the map when you're defining `my_layout`.

(continued)

16-8. Recent Earthquakes: You can find data files containing information about the most recent earthquakes over 1-hour, 1-day, 7-day, and 30-day periods online. Go to *https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php* and you'll see a list of links to data sets for various time periods, focusing on earthquakes of different magnitudes. Download one of these data sets, and create a visualization of the most recent earthquake activity.

16-9. World Fires: In the resources for this chapter, you'll find a file called *world_fires_1_day.csv*. This file contains information about fires burning in different locations around the globe, including the latitude and longitude, and the brightness of each fire. Using the data processing work from the first part of this chapter and the mapping work from this section, make a map that shows which parts of the world are affected by fires.

You can download more recent versions of this data at *https://earthdata.nasa.gov/earth-observation-data/near-real-time/firms/active-fire-data/*. You can find links to the data in CSV format in the *TXT* section.

Summary

In this chapter, you learned to work with real-world data sets. You processed CSV and JSON files, and extracted the data you want to focus on. Using historical weather data, you learned more about working with Matplotlib, including how to use the `datetime` module and how to plot multiple data series on one chart. You plotted geographical data on a world map in Plotly and styled Plotly maps and charts as well.

As you gain experience working with CSV and JSON files, you'll be able to process almost any data you want to analyze. You can download most online data sets in either or both of these formats. By working with these formats, you'll be able to learn how to work with other data formats more easily as well.

In the next chapter, you'll write programs that automatically gather their own data from online sources, and then you'll create visualizations of that data. These are fun skills to have if you want to program as a hobby and critical skills if you're interested in programming professionally.

17

WORKING WITH APIs

In this chapter, you'll learn how to write a self-contained program that generates a visualization based on data that it retrieves. Your program will use a web *application programming interface (API)* to automatically request specific information from a website—rather than entire pages—and then use that information to generate a visualization. Because programs written like this will always use current data to generate a visualization, even when that data might be rapidly changing, it will always be up to date.

Using a Web API

A web API is a part of a website designed to interact with programs. Those programs use very specific URLs to request certain information. This kind of request is called an *API call*. The requested data will be returned in an

easily processed format, such as JSON or CSV. Most apps that rely on external data sources, such as apps that integrate with social media sites, rely on API calls.

Git and GitHub

We'll base our visualization on information from GitHub, a site that allows programmers to collaborate on coding projects. We'll use GitHub's API to request information about Python projects on the site, and then generate an interactive visualization of the relative popularity of these projects using Plotly.

GitHub (*https://github.com/*) takes its name from Git, a distributed version control system. Git helps people manage their work on a project, so changes made by one person won't interfere with changes other people are making. When you implement a new feature in a project, Git tracks the changes you make to each file. When your new code works, you *commit* the changes you've made, and Git records the new state of your project. If you make a mistake and want to revert your changes, you can easily return to any previously working state. (To learn more about version control using Git, see Appendix D.) Projects on GitHub are stored in *repositories*, which contain everything associated with the project: its code, information on its collaborators, any issues or bug reports, and so on.

When users on GitHub like a project, they can "star" it to show their support and keep track of projects they might want to use. In this chapter, we'll write a program to automatically download information about the most-starred Python projects on GitHub, and then we'll create an informative visualization of these projects.

Requesting Data Using an API Call

GitHub's API lets you request a wide range of information through API calls. To see what an API call looks like, enter the following into your browser's address bar and press ENTER:

```
https://api.github.com/search/repositories?q=language:python&sort=stars
```

This call returns the number of Python projects currently hosted on GitHub, as well as information about the most popular Python repositories. Let's examine the call. The first part, `https://api.github.com/`, directs the request to the part of GitHub that responds to API calls. The next part, `search/repositories`, tells the API to conduct a search through all repositories on GitHub.

The question mark after `repositories` signals that we're about to pass an argument. The `q` stands for *query*, and the equal sign (`=`) lets us begin specifying a query (`q=`). By using `language:python`, we indicate that we want information only on repositories that have Python as the primary language. The final part, `&sort=stars`, sorts the projects by the number of stars they've been given.

The following snippet shows the first few lines of the response.

```
{
❶  "total_count": 3494012,
❷  "incomplete_results": false,
❸  "items": [
     {
       "id": 21289110,
       "node_id": "MDEwOlJlcG9zaXRvcnkyMTI4OTExMA==",
       "name": "awesome-python",
       "full_name": "vinta/awesome-python",
       --snip--
```

You can see from the response that this URL is not primarily intended to be entered by humans, because it's in a format that's meant to be processed by a program. GitHub found 3,494,012 Python projects as of this writing ❶. Because the value for `"incomplete_results"` is `false`, we know that the request was successful (it's not incomplete) ❷. If GitHub had been unable to fully process the API request, it would have returned `true` here. The `"items"` returned are displayed in the list that follows, which contains details about the most popular Python projects on GitHub ❸.

Installing Requests

The Requests package allows a Python program to easily request information from a website and examine the response. Use `pip` to install Requests:

```
$ python -m pip install --user requests
```

This line tells Python to run the `pip` module and install the Requests package to the current user's Python installation. If you use **`python3`** or a different command when running programs or installing packages, make sure you use the same command here.

NOTE *If this command doesn't work on macOS, try running the command again without the `--user` flag.*

Processing an API Response

Now we'll begin to write a program to automatically issue an API call and process the results by identifying the most starred Python projects on GitHub:

python_repos.py

```
❶ import requests

  # Make an API call and store the response.
❷ url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'
❸ headers = {'Accept': 'application/vnd.github.v3+json'}
❹ r = requests.get(url, headers=headers)
❺ print(f"Status code: {r.status_code}")
```

```
# Store API response in a variable.
❻ response_dict = r.json()

# Process results.
print(response_dict.keys())
```

At ❶ we import the `requests` module. At ❷ we store the URL of the API call in the `url` variable. GitHub is currently on the third version of its API, so we define headers for the API call ❸ that ask explicitly to use this version of the API. Then we use `requests` to make the call to the API ❹.

We call `get()` and pass it the URL and the header that we defined, and we assign the response object to the variable `r`. The response object has an attribute called `status_code`, which tells us whether the request was successful. (A status code of 200 indicates a successful response.) At ❺ we print the value of `status_code` so we can make sure the call went through successfully.

The API returns the information in JSON format, so we use the `json()` method to convert the information to a Python dictionary ❻. We store the resulting dictionary in `response_dict`.

Finally, we print the keys from `response_dict` and see this output:

```
Status code: 200
dict_keys(['total_count', 'incomplete_results', 'items'])
```

Because the status code is 200, we know that the request was successful. The response dictionary contains only three keys: `'total_count'`, `'incomplete_results'`, and `'items'`. Let's take a look inside the response dictionary.

NOTE *Simple calls like this should return a complete set of results, so it's safe to ignore the value associated with `'incomplete_results'`. But when you're making more complex API calls, your program should check this value.*

Working with the Response Dictionary

With the information from the API call stored as a dictionary, we can work with the data stored there. Let's generate some output that summarizes the information. This is a good way to make sure we received the information we expected and to start examining the information we're interested in:

python_repos.py

```
import requests

# Make an API call and store the response.
--snip--

# Store API response in a variable.
response_dict = r.json()
❶ print(f"Total repositories: {response_dict['total_count']}")

# Explore information about the repositories.
❷ repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")
```

```
# Examine the first repository.
❸ repo_dict = repo_dicts[0]
❹ print(f"\nKeys: {len(repo_dict)}")
❺ for key in sorted(repo_dict.keys()):
    print(key)
```

At ❶ we print the value associated with `'total_count'`, which represents the total number of Python repositories on GitHub.

The value associated with `'items'` is a list containing a number of dictionaries, each of which contains data about an individual Python repository. At ❷ we store this list of dictionaries in `repo_dicts`. We then print the length of `repo_dicts` to see how many repositories we have information for.

To look closer at the information returned about each repository, we pull out the first item from `repo_dicts` and store it in `repo_dict` ❸. We then print the number of keys in the dictionary to see how much information we have ❹. At ❺ we print all the dictionary's keys to see what kind of information is included.

The results give us a clearer picture of the actual data:

```
Status code: 200
Total repositories: 3494030
Repositories returned: 30

❶ Keys: 73
archive_url
archived
assignees_url
--snip--
url
watchers
watchers_count
```

GitHub's API returns a lot of information about each repository: there are 73 keys in `repo_dict` ❶. When you look through these keys, you'll get a sense of the kind of information you can extract about a project. (The only way to know what information is available through an API is to read the documentation or to examine the information through code, as we're doing here.)

Let's pull out the values for some of the keys in `repo_dict`:

python_repos.py

```
--snip--
# Explore information about the repositories.
repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")

# Examine the first repository.
repo_dict = repo_dicts[0]

print("\nSelected information about first repository:")
❶ print(f"Name: {repo_dict['name']}")
❷ print(f"Owner: {repo_dict['owner']['login']}")
❸ print(f"Stars: {repo_dict['stargazers_count']}")
print(f"Repository: {repo_dict['html_url']}")
```

```
❹ print(f"Created: {repo_dict['created_at']}")
❺ print(f"Updated: {repo_dict['updated_at']}")
  print(f"Description: {repo_dict['description']}")
```

Here, we print the values for a number of keys from the first repository's dictionary. At ❶ we print the name of the project. An entire dictionary represents the project's owner, so at ❷ we use the key `owner` to access the dictionary representing the owner, and then use the key `login` to get the owner's login name. At ❸ we print how many stars the project has earned and the URL for the project's GitHub repository. We then show when it was created ❹ and when it was last updated ❺. Finally, we print the repository's description; the output should look something like this:

```
Status code: 200
Total repositories: 3494032
Repositories returned: 30

Selected information about first repository:
Name: awesome-python
Owner: vinta
Stars: 61549
Repository: https://github.com/vinta/awesome-python
Created: 2014-06-27T21:00:06Z
Updated: 2019-02-17T04:30:00Z
Description: A curated list of awesome Python frameworks, libraries, software
    and resources
```

We can see that the most-starred Python project on GitHub as of this writing is *awesome-python*, its owner is user *vinta*, and it has been starred by more than 60,000 GitHub users. We can see the URL for the project's repository, its creation date of June 2014, and that it was updated recently. Additionally, the description tells us that awesome-python contains a list of popular Python resources.

Summarizing the Top Repositories

When we make a visualization for this data, we'll want to include more than one repository. Let's write a loop to print selected information about each repository the API call returns so we can include them all in the visualization:

python_repos.py

```
--snip--
# Explore information about the repositories.
repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")

❶ print("\nSelected information about each repository:")
❷ for repo_dict in repo_dicts:
    print(f"\nName: {repo_dict['name']}")
    print(f"Owner: {repo_dict['owner']['login']}")
    print(f"Stars: {repo_dict['stargazers_count']}")
    print(f"Repository: {repo_dict['html_url']}")
    print(f"Description: {repo_dict['description']}")
```

We print an introductory message at ❶. At ❷ we loop through all the dictionaries in `repo_dicts`. Inside the loop, we print the name of each project, its owner, how many stars it has, its URL on GitHub, and the project's description, as shown here:

```
Status code: 200
Total repositories: 3494040
Repositories returned: 30

Selected information about each repository:

Name: awesome-python
Owner: vinta
Stars: 61549
Repository: https://github.com/vinta/awesome-python
Description: A curated list of awesome Python frameworks, libraries, software
  and resources

Name: system-design-primer
Owner: donnemartin
Stars: 57256
Repository: https://github.com/donnemartin/system-design-primer
Description: Learn how to design large-scale systems. Prep for the system
  design interview.  Includes Anki flashcards.
--snip--

Name: python-patterns
Owner: faif
Stars: 19058
Repository: https://github.com/faif/python-patterns
Description: A collection of design patterns/idioms in Python
```

Some interesting projects appear in these results, and it might be worth looking at a few. But don't spend too much time, because shortly we'll create a visualization that will make the results much easier to read.

Monitoring API Rate Limits

Most APIs are rate limited, which means there's a limit to how many requests you can make in a certain amount of time. To see if you're approaching GitHub's limits, enter *https://api.github.com/rate_limit* into a web browser. You should see a response that begins like this:

```
{
  "resources": {
    "core": {
      "limit": 60,
      "remaining": 58,
      "reset": 1550385312
    },
❶   "search": {
❷     "limit": 10,
❸     "remaining": 8,
```

```
❹       "reset": 1550381772
      },
      --snip--
```

The information we're interested in is the rate limit for the search API ❶. We see at ❷ that the limit is 10 requests per minute and that we have 8 requests remaining for the current minute ❸. The reset value represents the time in *Unix* or *epoch time* (the number of seconds since midnight on January 1, 1970) when our quota will reset ❹. If you reach your quota, you'll get a short response that lets you know you've reached the API limit. If you reach the limit, just wait until your quota resets.

NOTE *Many APIs require you to register and obtain an API key to make API calls. As of this writing, GitHub has no such requirement, but if you obtain an API key, your limits will be much higher.*

Visualizing Repositories Using Plotly

Let's make a visualization using the data we have now to show the relative popularity of Python projects on GitHub. We'll make an interactive bar chart: the height of each bar will represent the number of stars the project has acquired, and you can click the bar's label to go to that project's home on GitHub. Save a copy of the program we've been working on as *python_repos_visual.py*, and then modify it so it reads as follows:

python_repos_visual.py

```
import requests

❶ from plotly.graph_objs import Bar
  from plotly import offline

❷ # Make an API call and store the response.
  url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'
  headers = {'Accept': 'application/vnd.github.v3+json'}
  r = requests.get(url, headers=headers)
  print(f"Status code: {r.status_code}")

  # Process results.
  response_dict = r.json()
  repo_dicts = response_dict['items']
❸ repo_names, stars = [], []
  for repo_dict in repo_dicts:
      repo_names.append(repo_dict['name'])
      stars.append(repo_dict['stargazers_count'])

  # Make visualization.
❹ data = [{
      'type': 'bar',
      'x': repo_names,
      'y': stars,
  }]
```

```
❺ my_layout = {
      'title': 'Most-Starred Python Projects on GitHub',
      'xaxis': {'title': 'Repository'},
      'yaxis': {'title': 'Stars'},
  }

  fig = {'data': data, 'layout': my_layout}
  offline.plot(fig, filename='python_repos.html')
```

We import the `Bar` class and the `offline` module from `plotly` ❶. We don't need to import the `Layout` class because we'll use the dictionary approach to define the layout, just as we did for the `data` list in the earthquake mapping project in Chapter 16. We continue to print the status of the API call response so we'll know if there is a problem ❷. We also remove some of the code that processes the API response, because we're no longer in the exploratory phase; we know we have the data we want.

We then create two empty lists ❸ to store the data we'll include in the initial chart. We'll need the name of each project to label the bars, and the number of stars to determine the height of the bars. In the loop, we append the name of each project and the number of stars it has to these lists.

Next, we define the `data` list ❹. This contains a dictionary, like we used in Chapter 16, which defines the type of the plot and provides the data for the x- and y-values. The x-values are the names of the projects, and the y-values are the number of stars each project has been given.

At ❺ we define the layout for this chart using the dictionary approach. Instead of making an instance of the `Layout` class, we build a dictionary with the layout specifications we want to use. We set a title for the overall chart, and we define a label for each axis.

Figure 17-1 shows the resulting chart. We can see that the first few projects are significantly more popular than the rest, but all of them are important projects in the Python ecosystem.

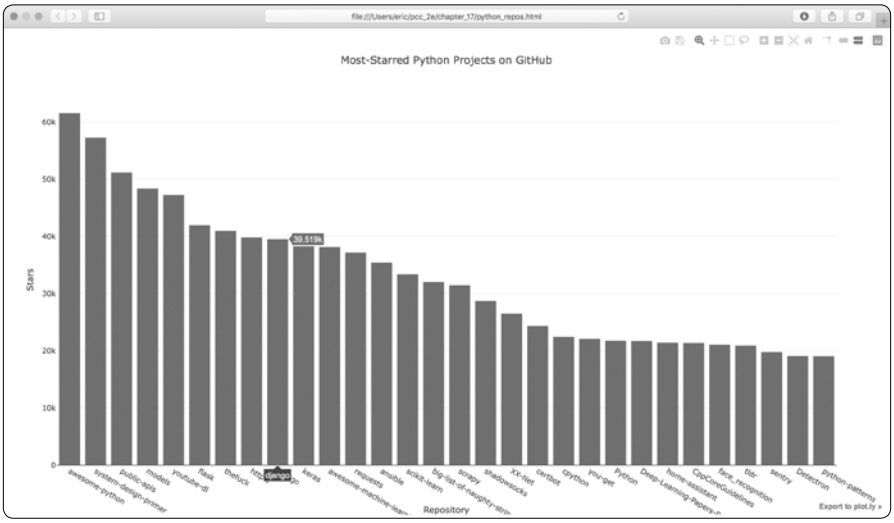


Figure 17-1: The most-starred Python projects on GitHub

Refining Plotly Charts

Let's refine the chart's styling. As you saw in Chapter 16, you can include all the styling directives as key-value pairs in the `data` and `my_layout` dictionaries.

Changes to the `data` object affect the bars. Here's a modified version of the `data` object for our chart that gives us a specific color and a clear border for each bar:

python_repos_visual.py

```
--snip--
data = [{
    'type': 'bar',
    'x': repo_names,
    'y': stars,
    'marker': {
        'color': 'rgb(60, 100, 150)',
        'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}
    },
    'opacity': 0.6,
}]
--snip--
```

The `marker` settings shown here affect the design of the bars. We set a custom blue color for the bars and specify that they'll be outlined with a dark gray line that's 1.5 pixels wide. We also set the opacity of the bars to 0.6 to soften the appearance of the chart a little.

Next, we'll modify `my_layout`:

python_repos_visual.py

```
--snip--
my_layout = {
    'title': 'Most-Starred Python Projects on GitHub',
❶   'titlefont': {'size': 28},
❷   'xaxis': {
        'title': 'Repository',
        'titlefont': {'size': 24},
        'tickfont': {'size': 14},
    },
❸   'yaxis': {
        'title': 'Stars',
        'titlefont': {'size': 24},
        'tickfont': {'size': 14},
    },
}
--snip--
```

We use the `'titlefont'` key to define the font size of the overall chart title ❶. Within the `'xaxis'` dictionary, we add settings to control the font size of the x-axis title (`'titlefont'`) and also of the tick labels (`'tickfont'`) ❷. Because these are individual nested dictionaries, you can include keys for the color and font family of the axis titles and tick labels. At ❸ we define similar settings for the y-axis.

Figure 17-2 shows the restyled chart.

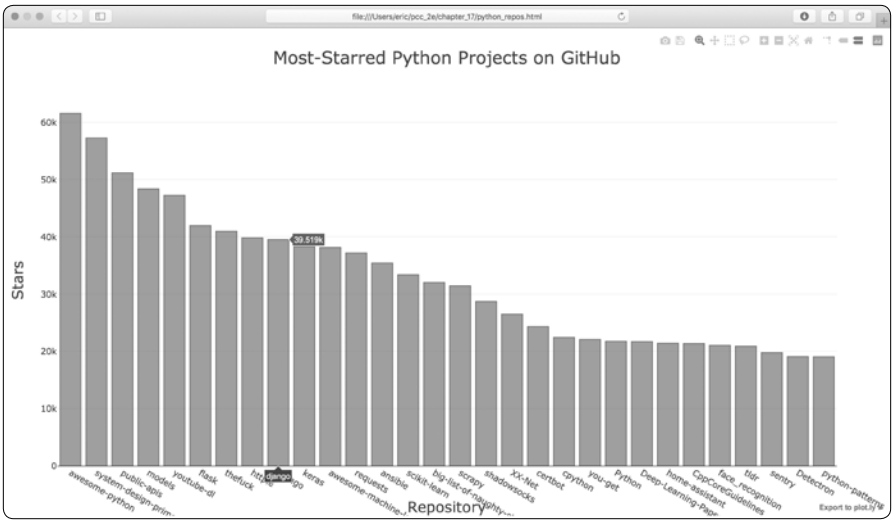


Figure 17-2: The styling for the chart has been refined.

Adding Custom Tooltips

In Plotly, you can hover the cursor over an individual bar to show the information that the bar represents. This is commonly called a *tooltip*, and in this case, it currently shows the number of stars a project has. Let's create a custom tooltip to show each project's description as well as the project's owner.

We need to pull some additional data to generate the tooltips and modify the `data` object:

python_repos_visual.py

```
--snip--
# Process results.
response_dict = r.json()
repo_dicts = response_dict['items']
❶ repo_names, stars, labels = [], [], []
for repo_dict in repo_dicts:
    repo_names.append(repo_dict['name'])
    stars.append(repo_dict['stargazers_count'])

❷     owner = repo_dict['owner']['login']
    description = repo_dict['description']
❸     label = f"{owner}<br />{description}"
    labels.append(label)


# Make visualization.
data = [{
    'type': 'bar',
    'x': repo_names,
    'y': stars,
❹     'hovertext': labels,
    'marker': {
```

```
        'color': 'rgb(60, 100, 150)',
        'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}
    },
    'opacity': 0.6,
}]
--snip--
```

We first define a new empty list, `labels`, to hold the text we want to display for each project ❶. In the loop where we process the data, we pull the owner and the description for each project ❷. Plotly allows you to use HTML code within text elements, so we generate a string for the label with a line break (`<br />`) between the project owner's username and the description ❸. We then store this label in the list `labels`.

In the `data` dictionary, we add an entry with the key `'hovertext'` and assign it the list we just created ❹. As Plotly creates each bar, it will pull labels from this list and only display them when the viewer hovers over a bar.

Figure 17-3 shows the resulting chart.

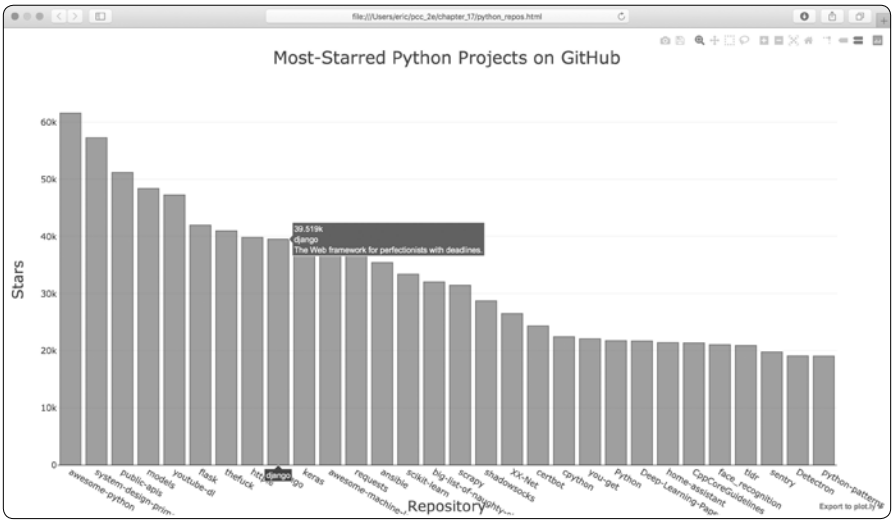


Figure 17-3: Hovering over a bar shows the project's owner and description.

Adding Clickable Links to Our Graph

Because Plotly allows you to use HTML on text elements, we can easily add links to a chart. Let's use the x-axis labels as a way to let the viewer visit any project's home page on GitHub. We need to pull the URLs from the data and use them when generating the x-axis labels:

python_repos_visual.py

```
--snip--
# Process results.
response_dict = r.json()
repo_dicts = response_dict['items']
```

```
❶ repo_links, stars, labels = [], [], []
  for repo_dict in repo_dicts:
      repo_name = repo_dict['name']
❷     repo_url = repo_dict['html_url']
❸     repo_link = f"<a href='{repo_url}'>{repo_name}</a>"
      repo_links.append(repo_link)

      stars.append(repo_dict['stargazers_count'])
      --snip--

  # Make visualization.
  data = [{
      'type': 'bar',
❹     'x': repo_links,
      'y': stars,
      --snip--
  }]
  --snip--
```

We update the name of the list we're creating from `repo_names` to `repo_links` to more accurately communicate the kind of information we're putting together for the chart ❶. We then pull the URL for the project from `repo_dict` and assign it to the temporary variable `repo_url` ❷. At ❸ we generate a link to the project. We use the HTML anchor tag, which has the form `<a href='URL'>link text</a>`, to generate the link. We then append this link to the list `repo_links`.

At ❹ we use this list for the x-values in the chart. The result looks the same as before, but now the viewer can click any of the project names at the bottom of the chart to visit that project's home page on GitHub. Now we have an interactive, informative visualization of data retrieved through an API!

More About Plotly and the GitHub API

To read more about working with Plotly charts, there are two good places to start. You can find the *Plotly User Guide in Python* at *https://plot.ly/python/user-guide/*. This resource gives you a better understanding of how Plotly uses your data to construct a visualization and why it approaches defining data visualizations in this way.

The *python figure reference* at *https://plot.ly/python/reference/* lists all the settings you can use to configure Plotly visualizations. All the possible chart types are listed as well as all the attributes you can set for every configuration option.

For more about the GitHub API, refer to its documentation at *https://developer.github.com/v3/*. Here you'll learn how to pull a wide variety of specific information from GitHub. If you have a GitHub account, you can work with your own data as well as the publicly available data for other users' repositories.

The Hacker News API

To explore how to use API calls on other sites, let's take a quick look at Hacker News (*http://news.ycombinator.com/*). On Hacker News, people share articles about programming and technology, and engage in lively discussions about those articles. The Hacker News API provides access to data about all submissions and comments on the site, and you can use the API without having to register for a key.

The following call returns information about the current top article as of this writing:

```
https://hacker-news.firebaseio.com/v0/item/19155826.json
```

When you enter this URL in a browser, you'll see that the text on the page is enclosed by braces, meaning it's a dictionary. But the response is difficult to examine without some better formatting. Let's run this URL through the `json.dump()` method, like we did in the earthquake project in Chapter 16, so we can explore the kind of information that's returned about an article:

hn_article.py

```
import requests
import json

# Make an API call, and store the response.
url = 'https://hacker-news.firebaseio.com/v0/item/19155826.json'
r = requests.get(url)
print(f"Status code: {r.status_code}")

# Explore the structure of the data.
response_dict = r.json()
readable_file = 'data/readable_hn_data.json'
with open(readable_file, 'w') as f:
    json.dump(response_dict, f, indent=4)
```

Everything in this program should look familiar, because we've used it all in the previous two chapters. The output is a dictionary of information about the article with the ID 19155826:

readable_hn_data.json

```
{
    "by": "jimktrains2",
❶   "descendants": 220,
    "id": 19155826,
❷   "kids": [
        19156572,
        19158857,
        --snip--
    ],
    "score": 722,
    "time": 1550085414,
❸   "title": "Nasa's Mars Rover Opportunity Concludes a 15-Year Mission",
    "type": "story",
❹   "url": "https://www.nytimes.com/.../mars-opportunity-rover-dead.html"
}
```

The dictionary contains a number of keys we can work with. The key `'descendants'` tells us the number of comments the article has received ❶. The key `'kids'` provides the IDs of all comments made directly in response to this submission ❷. Each of these comments might have comments of their own as well, so the number of descendants a submission has is usually greater than its number of kids. We can see the title of the article being discussed ❸, and a URL for the article that's being discussed as well ❹.

The following URL returns a simple list of all the IDs of the current top articles on Hacker News:

```
https://hacker-news.firebaseio.com/v0/topstories.json
```

We can use this call to find out which articles are on the home page right now, and then generate a series of API calls similar to the one we just examined. With this approach, we can print a summary of all the articles on the front page of Hacker News at the moment:

hn_submissions.py

```
from operator import itemgetter

import requests

# Make an API call and store the response.
❶ url = 'https://hacker-news.firebaseio.com/v0/topstories.json'
r = requests.get(url)
print(f"Status code: {r.status_code}")

# Process information about each submission.
❷ submission_ids = r.json()
❸ submission_dicts = []
for submission_id in submission_ids[:30]:
    # Make a separate API call for each submission.
❹     url = f"https://hacker-news.firebaseio.com/v0/item/{submission_id}.json"
    r = requests.get(url)
    print(f"id: {submission_id}\tstatus: {r.status_code}")
    response_dict = r.json()

    # Build a dictionary for each article.
❺     submission_dict = {
        'title': response_dict['title'],
        'hn_link': f"http://news.ycombinator.com/item?id={submission_id}",
        'comments': response_dict['descendants'],
    }
❻     submission_dicts.append(submission_dict)

❼ submission_dicts = sorted(submission_dicts, key=itemgetter('comments'),
                            reverse=True)

❽ for submission_dict in submission_dicts:
    print(f"\nTitle: {submission_dict['title']}")
    print(f"Discussion link: {submission_dict['hn_link']}")
    print(f"Comments: {submission_dict['comments']}")
```

First, we make an API call, and then print the status of the response ❶. This API call returns a list containing the IDs of up to the 500 most popular articles on Hacker News at the time the call is issued. We then convert the response object to a Python list at ❷, which we store in `submission_ids`. We'll use these IDs to build a set of dictionaries that each store information about one of the current submissions.

We set up an empty list called `submission_dicts` at ❸ to store these dictionaries. We then loop through the IDs of the top 30 submissions. We make a new API call for each submission by generating a URL that includes the current value of `submission_id` ❹. We print the status of each request along with its ID, so we can see whether it's successful.

At ❺ we create a dictionary for the submission currently being processed, where we store the title of the submission, a link to the discussion page for that item, and the number of comments the article has received so far. Then we append each `submission_dict` to the list `submission_dicts` ❻.

Each submission on Hacker News is ranked according to an overall score based on a number of factors including how many times it's been voted up, how many comments it's received, and how recent the submission is. We want to sort the list of dictionaries by the number of comments. To do this, we use a function called `itemgetter()` ❼, which comes from the `operator` module. We pass this function the key `'comments'`, and it pulls the value associated with that key from each dictionary in the list. The `sorted()` function then uses this value as its basis for sorting the list. We sort the list in reverse order to place the most-commented stories first.

Once the list is sorted, we loop through the list at ❽ and print out three pieces of information about each of the top submissions: the title, a link to the discussion page, and the number of comments the submission currently has:

```
Status code: 200
id: 19155826    status: 200
id: 19180181    status: 200
id: 19181473    status: 200
--snip--

Title: Nasa's Mars Rover Opportunity Concludes a 15-Year Mission
Discussion link: http://news.ycombinator.com/item?id=19155826
Comments: 220

Title: Ask HN: Is it practical to create a software-controlled model rocket?
Discussion link: http://news.ycombinator.com/item?id=19180181
Comments: 72

Title: Making My Own USB Keyboard from Scratch
Discussion link: http://news.ycombinator.com/item?id=19181473
Comments: 62
--snip--
```

You would use a similar process to access and analyze information with any API. With this data, you could make a visualization showing which submissions have inspired the most active recent discussions. This is also the basis for apps that provide a customized reading experience for sites like Hacker News. To learn more about what kind of information you can access through the Hacker News API, visit the documentation page at *https://github.com/HackerNews/API/*.

TRY IT YOURSELF

17-1. Other Languages: Modify the API call in *python_repos.py* so it generates a chart showing the most popular projects in other languages. Try languages such as *JavaScript*, *Ruby*, *C*, *Java*, *Perl*, *Haskell*, and *Go*.

17-2. Active Discussions: Using the data from *hn_submissions.py*, make a bar chart showing the most active discussions currently happening on Hacker News. The height of each bar should correspond to the number of comments each submission has. The label for each bar should include the submission's title and should act as a link to the discussion page for that submission.

17-3. Testing python_repos.py: In *python_repos.py*, we printed the value of `status_code` to make sure the API call was successful. Write a program called *test_python_repos.py* that uses `unittest` to assert that the value of `status_code` is 200. Figure out some other assertions you can make—for example, that the number of items returned is expected and that the total number of repositories is greater than a certain amount.

17-4. Further Exploration: Visit the documentation for Plotly and either the GitHub API or the Hacker News API. Use some of the information you find there to either customize the style of the plots we've already made or pull some different information and create your own visualizations.

Summary

In this chapter, you learned how to use APIs to write self-contained programs that automatically gather the data they need and use that data to create a visualization. You used the GitHub API to explore the most-starred Python projects on GitHub, and you also looked briefly at the Hacker News API. You learned how to use the Requests package to automatically issue an API call to GitHub and how to process the results of that call. Some Plotly settings were also introduced that further customize the appearance of the charts you generate.

In the next chapter, you'll use Django to build a web application as your final project.