6

DICTIONARIES

In this chapter you'll learn how to use Python's dictionaries, which allow you to connect pieces of related information. You'll learn how to access the information once it's in a dictionary and how to modify that information. Because dictionaries can store an almost limitless amount of information, I'll show you how to loop through the data in a dictionary. Additionally, you'll learn to nest dictionaries inside lists, lists inside dictionaries, and even dictionaries inside other dictionaries.

Understanding dictionaries allows you to model a variety of real-world objects more accurately. You'll be able to create a dictionary representing a person and then store as much information as you want about that person. You can store their name, age, location, profession, and any other aspect of a person you can describe. You'll be able to store any two kinds of

information that can be matched up, such as a list of words and their meanings, a list of people's names and their favorite numbers, a list of mountains and their elevations, and so forth.

A Simple Dictionary

Consider a game featuring aliens that can have different colors and point values. This simple dictionary stores information about a particular alien:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

The dictionary `alien_0` stores the alien's color and point value. The last two lines access and display that information, as shown here:

```
green
5
```

As with most new programming concepts, using dictionaries takes practice. Once you've worked with dictionaries for a bit you'll soon see how effectively they can model real-world situations.

Working with Dictionaries

A *dictionary* in Python is a collection of *key-value pairs*. Each *key* is connected to a value, and you can use a key to access the value associated with that key. A key's value can be a number, a string, a list, or even another dictionary. In fact, you can use any object that you can create in Python as a value in a dictionary.

In Python, a dictionary is wrapped in braces, {}, with a series of key-value pairs inside the braces, as shown in the earlier example:

```
alien_0 = {'color': 'green', 'points': 5}
```

A *key-value pair* is a set of values associated with each other. When you provide a key, Python returns the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas. You can store as many key-value pairs as you want in a dictionary.

The simplest dictionary has exactly one key-value pair, as shown in this modified version of the `alien_0` dictionary:

```
alien_0 = {'color': 'green'}
```

This dictionary stores one piece of information about `alien_0`, namely the alien's color. The string `'color'` is a key in this dictionary, and its associated value is `'green'`.

Accessing Values in a Dictionary

To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets, as shown here:

alien.py

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

This returns the value associated with the key `'color'` from the dictionary `alien_0`:

```
green
```

You can have an unlimited number of key-value pairs in a dictionary. For example, here's the original `alien_0` dictionary with two key-value pairs:

```
alien_0 = {'color': 'green', 'points': 5}
```

Now you can access either the color or the point value of `alien_0`. If a player shoots down this alien, you can look up how many points they should earn using code like this:

```
alien_0 = {'color': 'green', 'points': 5}

❶ new_points = alien_0['points']
❷ print(f"You just earned {new_points} points!")
```

Once the dictionary has been defined, the code at ❶ pulls the value associated with the key `'points'` from the dictionary. This value is then assigned to the variable `new_points`. The line at ❷ prints a statement about how many points the player just earned:

```
You just earned 5 points!
```

If you run this code every time an alien is shot down, the alien's point value will be retrieved.

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. For example, to add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets along with the new value.

Let's add two new pieces of information to the `alien_0` dictionary: the alien's x- and y-coordinates, which will help us display the alien in a

particular position on the screen. Let's place the alien on the left edge of the screen, 25 pixels down from the top. Because screen coordinates usually start at the upper-left corner of the screen, we'll place the alien on the left edge of the screen by setting the x-coordinate to 0 and 25 pixels from the top by setting its y-coordinate to positive 25, as shown here:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

❶ alien_0['x_position'] = 0
❷ alien_0['y_position'] = 25
print(alien_0)
```

We start by defining the same dictionary that we've been working with. We then print this dictionary, displaying a snapshot of its information. At ❶ we add a new key-value pair to the dictionary: key `'x_position'` and value `0`. We do the same for key `'y_position'` at ❷. When we print the modified dictionary, we see the two additional key-value pairs:

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

The final version of the dictionary contains four key-value pairs. The original two specify color and point value, and two more specify the alien's position.

NOTE *As of Python 3.7, dictionaries retain the order in which they were defined. When you print a dictionary or loop through its elements, you will see the elements in the same order in which they were added to the dictionary.*

Starting with an Empty Dictionary

It's sometimes convenient, or even necessary, to start with an empty dictionary and then add each new item to it. To start filling an empty dictionary, define a dictionary with an empty set of braces and then add each key-value pair on its own line. For example, here's how to build the `alien_0` dictionary using this approach:

alien.py

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Here we define an empty `alien_0` dictionary, and then add color and point values to it. The result is the dictionary we've been using in previous examples:

```
{'color': 'green', 'points': 5}
```

Typically, you'll use empty dictionaries when storing user-supplied data in a dictionary or when you write code that generates a large number of key-value pairs automatically.

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. For example, consider an alien that changes from green to yellow as a game progresses:

alien.py

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")

alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.")
```

We first define a dictionary for `alien_0` that contains only the alien's color; then we change the value associated with the key `'color'` to `'yellow'`. The output shows that the alien has indeed changed from green to yellow:

```
The alien is green.
The alien is now yellow.
```

For a more interesting example, let's track the position of an alien that can move at different speeds. We'll store a value representing the alien's current speed and then use it to determine how far to the right the alien should move:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
print(f"Original position: {alien_0['x_position']}")

# Move the alien to the right.
# Determine how far to move the alien based on its current speed.
❶ if alien_0['speed'] == 'slow':
    x_increment = 1
elif alien_0['speed'] == 'medium':
    x_increment = 2
else:
    # This must be a fast alien.
    x_increment = 3

# The new position is the old position plus the increment.
❷ alien_0['x_position'] = alien_0['x_position'] + x_increment

print(f"New position: {alien_0['x_position']}")
```

We start by defining an alien with an initial *x* position and *y* position, and a speed of `'medium'`. We've omitted the color and point values for the

sake of simplicity, but this example would work the same way if you included those key-value pairs as well. We also print the original value of `x_position` to see how far the alien moves to the right.

At ❶, an `if-elif-else` chain determines how far the alien should move to the right and assigns this value to the variable `x_increment`. If the alien's speed is `'slow'`, it moves one unit to the right; if the speed is `'medium'`, it moves two units to the right; and if it's `'fast'`, it moves three units to the right. Once the increment has been calculated, it's added to the value of `x_position` at ❷, and the result is stored in the dictionary's `x_position`.

Because this is a medium-speed alien, its position shifts two units to the right:

```
Original x-position: 0
New x-position: 2
```

This technique is pretty cool: by changing one value in the alien's dictionary, you can change the overall behavior of the alien. For example, to turn this medium-speed alien into a fast alien, you would add the line:

```
alien_0['speed'] = 'fast'
```

The `if-elif-else` block would then assign a larger value to `x_increment` the next time the code runs.

Removing Key-Value Pairs

When you no longer need a piece of information that's stored in a dictionary, you can use the `del` statement to completely remove a key-value pair. All `del` needs is the name of the dictionary and the key that you want to remove.

For example, let's remove the key `'points'` from the `alien_0` dictionary along with its value:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

❶ del alien_0['points']
print(alien_0)
```

The line at ❶ tells Python to delete the key `'points'` from the dictionary `alien_0` and to remove the value associated with that key as well. The output shows that the key `'points'` and its value of `5` are deleted from the dictionary, but the rest of the dictionary is unaffected:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

NOTE *Be aware that the deleted key-value pair is removed permanently.*

A Dictionary of Similar Objects

The previous example involved storing different kinds of information about one object, an alien in a game. You can also use a dictionary to store one kind of information about many objects. For example, say you want to poll a number of people and ask them what their favorite programming language is. A dictionary is useful for storing the results of a simple poll, like this:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }
```

As you can see, we've broken a larger dictionary into several lines. Each key is the name of a person who responded to the poll, and each value is their language choice. When you know you'll need more than one line to define a dictionary, press ENTER after the opening brace. Then indent the next line one level (four spaces), and write the first key-value pair, followed by a comma. From this point forward when you press ENTER, your text editor should automatically indent all subsequent key-value pairs to match the first key-value pair.

Once you've finished defining the dictionary, add a closing brace on a new line after the last key-value pair and indent it one level so it aligns with the keys in the dictionary. It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line.

NOTE *Most editors have some functionality that helps you format extended lists and dictionaries in a similar manner to this example. Other acceptable ways to format long dictionaries are available as well, so you may see slightly different formatting in your editor, or in other sources.*

To use this dictionary, given the name of a person who took the poll, you can easily look up their favorite language:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

❶ language = favorite_languages['sarah'].title()
print(f"Sarah's favorite language is {language}.")
```

To see which language Sarah chose, we ask for the value at:

```
favorite_languages['sarah']
```

We use this syntax to pull Sarah's favorite language from the dictionary at ❶ and assign it to the variable `language`. Creating a new variable here makes for a much cleaner `print()` call. The output shows Sarah's favorite language:

```
Sarah's favorite language is C.
```

You could use this same syntax with any individual represented in the dictionary.

Using get() to Access Values

Using keys in square brackets to retrieve the value you're interested in from a dictionary might cause one potential problem: if the key you ask for doesn't exist, you'll get an error.

Let's see what happens when you ask for the point value of an alien that doesn't have a point value set:

alien_no_points.py

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0['points'])
```

This results in a traceback, showing a `KeyError`:

```
Traceback (most recent call last):
  File "alien_no_points.py", line 2, in <module>
    print(alien_0['points'])
KeyError: 'points'
```

You'll learn more about how to handle errors like this in general in Chapter 10. For dictionaries, specifically, you can use the `get()` method to set a default value that will be returned if the requested key doesn't exist.

The `get()` method requires a key as a first argument. As a second optional argument, you can pass the value to be returned if the key doesn't exist:

```
alien_0 = {'color': 'green', 'speed': 'slow'}

point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

If the key `'points'` exists in the dictionary, you'll get the corresponding value. If it doesn't, you get the default value. In this case, `points` doesn't exist, and we get a clean message instead of an error:

```
No point value assigned.
```

If there's a chance the key you're asking for might not exist, consider using the `get()` method instead of the square bracket notation.

NOTE *If you leave out the second argument in the call to* `get()` *and the key doesn't exist, Python will return the value* `None`*. The special value* `None` *means "no value exists." This is not an error: it's a special value meant to indicate the absence of a value. You'll see more uses for* `None` *in Chapter 8.*

TRY IT YOURSELF

6-1. Person: Use a dictionary to store information about a person you know. Store their first name, last name, age, and the city in which they live. You should have keys such as `first_name`, `last_name`, `age`, and `city`. Print each piece of information stored in your dictionary.

6-2. Favorite Numbers: Use a dictionary to store people's favorite numbers. Think of five names, and use them as keys in your dictionary. Think of a favorite number for each person, and store each as a value in your dictionary. Print each person's name and their favorite number. For even more fun, poll a few friends and get some actual data for your program.

6-3. Glossary: A Python dictionary can be used to model an actual dictionary. However, to avoid confusion, let's call it a glossary.

- Think of five programming words you've learned about in the previous chapters. Use these words as the keys in your glossary, and store their meanings as values.
- Print each word and its meaning as neatly formatted output. You might print the word followed by a colon and then its meaning, or print the word on one line and then print its meaning indented on a second line. Use the newline character (`\n`) to insert a blank line between each word-meaning pair in your output.

Looping Through a Dictionary

A single Python dictionary can contain just a few key-value pairs or millions of pairs. Because a dictionary can contain large amounts of data, Python lets you loop through a dictionary. Dictionaries can be used to store information in a variety of ways; therefore, several different ways exist to loop through them. You can loop through all of a dictionary's key-value pairs, through its keys, or through its values.

Looping Through All Key-Value Pairs

Before we explore the different approaches to looping, let's consider a new dictionary designed to store information about a user on a website. The