Chapter 6 Dictionaries

6.1 A Simple Dictionary

6.2 Working with Dictionaries

6.3 Looping Through a Dictionary

6.4 Nesting

6.5 Summary

6.1 A Simple Dictionary

This simple dictionary stores information about a particular alien:

```
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 two print statements access and display that information, as shown here:

```
green
5
```

6.2 Working with Dictionaries

A *dictionary* in Python is a collection of *key-value pairs*.
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}
```

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'}
```

6.2.1 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_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("You just earned " + str(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 stored in the variable new_points. The line at ❷ converts this integer value to a string and prints a statement about how many points the player just earned:

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

6.2.2 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_0 = {'color': 'green', 'points': 5}
print(alien_0)
❶ alien_0['x_position'] = 0
❷ alien_0['y_position'] = 25
print(alien_0)
```

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}
```

6.2.3 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_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}
```

6.2.4 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_0 = {'color': 'green'}
print("The alien is " + alien_0['color'] + ".")
alien_0['color'] = 'yellow'
print("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("Original x-position: " + str(alien_0['x_position']))
# Move the alien to the right.
# Determine how far to move the alien based on its current speed.
❶if 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.
❷alien_0['x_position'] = alien_0['x_position'] + x_increment
print("New x-position: " + str(alien_0['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'
```

6.2.5 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_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.*

6.2.6 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',
}
```

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.

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

❶print("Sarah's favorite language is " +
❷       favorite_languages['sarah'].title() +
❸       ".")
```

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

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

This syntax is used in the `print` statement at ❷, and the output shows Sarah's favorite language:

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

This example also shows how you can break up a long `print` statement over several lines. The word `print` is shorter than most dictionary names, so it makes sense to include the first part of what you want to print right after the opening parenthesis ❶. Choose an appropriate point at which to break what's being printed, and add a concatenation operator (+) at the end of the first line ❷. Press ENTER and then press TAB to align all subsequent lines at one indentation level under the `print` statement. When you've finished composing your output, you can place the closing parenthesis on the last line of the `print` block ❸.

6.3 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.

6.3.1 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 following dictionary would store one person's username, first name, and last name:

```
user_0 = {
        'username': 'efermi',
        'first': 'enrico',
        'last': 'fermi',
        }
```

You can access any single piece of information about `user_0` based on what you've already learned in this chapter. But what if you wanted to see everything stored in this user's dictionary? To do so, you could loop through the dictionary using a `for` loop:

```
user_0 = {
        'username': 'efermi',
        'first': 'enrico',
        'last': 'fermi',
        }
❶ for key, value in user_0.items():
❷       print("\nKey: " + key)
❸       print("Value: " + value)
```

As shown at ❶, to write a `for` loop for a dictionary, you create names for the two variables that will hold the key and value in each key-value pair. You can choose any names you want for these two variables. This code would work just as well if you had used abbreviations for the variable names, like this:

```
for k, v in user_0.items()
```

The `"\n"` in the first `print` statement ensures that a blank line is inserted before each key-value pair in the output:

```
Key: last
Value: fermi

Key: first
Value: enrico

Key: username
Value: efermi
```

Looping through all key-value pairs works particularly well for dictionaries like the *favorite_languages.py* example on 6.2.6, which stores the same kind of information for many different keys. If you loop through the `favorite_languages` dictionary, you get the name of each person in the dictionary and their favorite programming language. Because the keys always refer to a person's name and the value is always a language, we'll use the variables `name` and `language` in the loop instead of `key` and `value`. This will make it easier to follow what's happening inside the loop:

```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }
❶ for name, language in favorite_languages.items():
❷       print(name.title() + "'s favorite language is " +
                language.title() + ".")
```

Now, in just a few lines of code, we can display all of the information from the poll:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Phil's favorite language is Python.
Edward's favorite language is Ruby.
```

6.3.2 Looping Through All the Keys in a Dictionary

The `keys()` method is useful when you don't need to work with all of the values in a dictionary. Let's loop through the `favorite_languages` dictionary and print the names of everyone who took the poll:

```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }
❶ for name in favorite_languages.keys():
        print(name.title())
```

The line at ❶ tells Python to pull all the keys from the dictionary `favorite_languages` and store them one at a time in the variable name. The output shows the names of everyone who took the poll:

```
Jen
Sarah
Phil
Edward
```

Looping through the keys is actually the default behavior when looping through a dictionary, so this code would have exactly the same output if you wrote . . .

```
for name in favorite_languages:
```

rather than...

```
for name in favorite_languages.keys():
```

You can access the value associated with any key you care about inside the loop by using the current key. Let's print a message to a couple of friends about the languages they chose. We'll loop through the names in the dictionary as we did previously, but when the name matches one of our friends, we'll display a message about their favorite language:

```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }
❶ friends = ['phil', 'sarah']
for name in favorite_languages.keys():
        print(name.title())

❷       if name in friends:
                print(" Hi " + name.title() +
                ", I see your favorite language is " +
❸               favorite_languages[name].title() + "!")
```

At ❶ we make a list of friends that we want to print a message to. Inside the loop, we print each person's name. Then at ❷ we check to see whether the `name` we are working with is in the list friends. If it is, we print a special greeting, including a reference to their language choice. To access the favorite language at ❸, we use the name of the dictionary and the current value of `name` as the key. Everyone's name is printed, but our friends receive a special message:

```
Edward
Phil
        Hi Phil, I see your favorite language is Python!
Sarah
        Hi Sarah, I see your favorite language is C!
Jen
```

You can also use the `keys()` method to find out if a particular person was polled. This time, let's find out if Erin took the poll:

```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }
❶ if 'erin' not in favorite_languages.keys():
        print("Erin, please take our poll!")
```

The `keys()` method isn't just for looping: It actually returns a list of all the keys, and the line at ❶ simply checks if `'erin'` is in this list. Because she's not, a message is printed inviting her to take the poll:

```
Erin, please take our poll!
```

6.3.4 Looping Through All Values in a Dictionary

If you are primarily interested in the values that a dictionary contains, you can use the `values()` method to return a list of values without any keys. For example, say we simply want a list of all languages chosen in our programming language poll without the name of the person who chose each language:

```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }
print("The following languages have been mentioned:")
for language in favorite_languages.values():
        print(language.title())
```

The `for` statement here pulls each value from the dictionary and stores it in the variable language. When these values are printed, we get a list of all chosen languages:

```
The following languages have been mentioned:
Python
C
Python
Ruby
```

This approach pulls all the values from the dictionary without checking for repeats. That might work fine with a small number of values, but in a poll with a large number of respondents, this would result in a very repetitive list. To see each language chosen without repetition, we can use a set.

A set is similar to a list except that each item in the set must be unique:

```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }
print("The following languages have been mentioned:")
❶ for language in set(favorite_languages.values()):
        print(language.title())
```

When you wrap `set()` around a list that contains duplicate items, Python identifies the unique items in the list and builds a set from those items. At ❶ we use `set()` to pull out the unique languages in `favorite_languages`.values().
The result is a nonrepetitive list of languages that have been mentioned by people taking the poll:

```
The following languages have been mentioned:
Python
C
Ruby
```

6.4 Nesting

Sometimes you'll want to store a set of dictionaries in a list or a list of items as a value in a dictionary. This is called nesting. You can nest a set of dictionaries inside a list, a list of items inside a dictionary, or even a dictionary inside another dictionary. Nesting is a powerful feature, as the following examples will demonstrate.

6.4.1 A List of Dictionaries

The `alien_0` dictionary contains a variety of information about one alien, but it has no room to store information about a second alien, much less a screen full of aliens. How can you manage a fleet of aliens? One way is to make a list of aliens in which each alien is a dictionary of information about that alien. For example, the following code builds a list of three aliens:

```
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}

❶ aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
print(alien)
```

We first create three dictionaries, each representing a different alien. At ❶ we pack each of these dictionaries into a list called aliens. Finally, we loop through the list and print out each alien:

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

A more realistic example would involve more than three aliens with code that automatically generates each alien. In the following example we use `range()` to create a fleet of 30 aliens:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
❶ for alien_number in range(30):
❷       new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
❸       aliens.append(new_alien)
# Show the first 5 aliens:
❹ for alien in aliens[:5]:
        print(alien)
print("...")
# Show how many aliens have been created.
❺ print("Total number of aliens: " + str(len(aliens)))
```

This example begins with an empty list to hold all of the aliens that will be created. At ❶ `range()` returns a set of numbers, which just tells Python how many times we want the loop to repeat. Each time the loop runs we create a new alien ❷ and then append each new alien to the list `aliens` ❸. At ❹ we use a slice to print the first five aliens, and then at ❺ we print the length of the list to prove we've actually generated the full fleet of 30 aliens:

```
{'speed':'slow','color':'green','points':5}
{'speed':'slow','color':'green','points':5}
{'speed':'slow','color':'green','points':5}
{'speed':'slow','color':'green','points':5}
{'speed':'slow','color':'green','points':5}
...
Total number of aliens: 30
```

How might you work with a set of aliens like this? Imagine that one aspect of a game has some aliens changing color and moving faster as the game progresses. When it's time to change colors, we can use a `for` loop and an `if` statement to change the color of aliens. For example, to change the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

```
# Make an empty list for storing aliens.
aliens = []
# Make 30 green aliens.
for alien_number in range (0,30):
        new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
        aliens.append(new_alien)
for alien in aliens[0:3]:
        if alien['color'] == 'green':
                alien['color'] = 'yellow'
                alien['speed'] = 'medium'
                alien['points'] = 10
# Show the first 5 aliens:
for alien in aliens[0:5]:
        print(alien)
print("...")
```

Because we want to modify the first three aliens, we loop through a slice that includes only the first three aliens. All of the aliens are green now but that won't always be the case, so we write an `if` statement to make sure we're only modifying green aliens. If the alien is green, we change the color to `'yellow'`, the speed to `'medium'`, and the point value to 10, as shown in the following output:

```
{'speed':'slow','color':'green','points':10}
{'speed':'slow','color':'green','points':10}
{'speed':'slow','color':'green','points':10}
{'speed':'slow','color':'green','points':5}
{'speed':'slow','color':'green','points':5}
...
```

You could expand this loop by adding an `elif` block that turns yellow aliens into red, fast-moving ones worth 15 points each. Without showing the entire program again, that loop would look like this:

```
for alien in aliens[0:3]:
        if alien['color'] == 'green':
                alien['color'] = 'yellow'
                alien['speed'] = 'medium'
                alien['points'] = 10
        elif alien['color'] == 'yellow':
                alien['color'] = 'red'
                alien['speed'] = 'fast'
                alien['points'] = 15
```

6.4.2 A List in a Dictionary

In the following example, two kinds of information are stored for each pizza: a type of crust and a list of toppings. The list of toppings is a value associated with the key `'toppings'`. To use the items in the list, we give the name of the dictionary and the key `'toppings'`, as we would any value in the dictionary. Instead of returning a single value, we get a list of toppings:

```
# Store information about a pizza being ordered.
pizza = {
        'crust': 'thick',
        'toppings': ['mushrooms', 'extra cheese'],
        }
# Summarize the order.
print("You ordered a " + pizza['crust'] + "-crust pizza " +
        "with the following toppings:")

for topping in pizza['toppings']:
        print("\t" + topping)
```

The following output summarizes the pizza that we plan to build:

```
You ordered a thick-crust pizza with the following toppings:
        mushrooms
        extra cheese
```

You can nest a list inside a dictionary any time you want more than one value to be associated with a single key in a dictionary. In the earlier example of favorite programming languages, if we were to store each person's responses in a list, people could choose more than one favorite language. When we loop through the dictionary, the value associated with each person would be a list of languages rather than a single language. Inside the dictionary's `for` loop, we use another `for` loop to run through the list of languages associated with each person:

```
❶ favorite_languages = {
        'jen': ['python', 'ruby'],
        'sarah': ['c'],
        'edward': ['ruby', 'go'],
        'phil': ['python', 'haskell'],
        }
❷ for name, languages in favorite_languages.items():
        print("\n" + name.title() + "'s favorite languages are:")
❸       for language in languages:
                print("\t" + language.title())
```

As you can see at ❶ the value associated with each name is now a list. Notice that some people have one favorite language and others have multiple favorites. When we loop through the dictionary at ❷, we use the variable name `languages` to hold each value from the dictionary, because we know that each value will be a list. Inside the main dictionary loop, we use another `for` loop ❸ to run through each person's list of favorite languages. Now each person can list as many favorite languages as they like:

```
Jen's favorite languages are:
        Python
        Ruby
Sarah's favorite languages are:
        C
Phil's favorite languages are:
        Python
        Haskell
Edward's favorite languages are:
        Ruby
        Go
```

6.4.3 A Dictionary in a Dictionary

You can nest a dictionary inside another dictionary, but your code can get complicated quickly when you do. For example, if you have several users for a website, each with a unique username, you can use the usernames as the keys in a dictionary. You can then store information about each user by using a dictionary as the value associated with their username. In the following listing, we store three pieces of information about each user: their first name, last name, and location. We'll access this information by looping through the usernames and the dictionary of information associated with each username:

```
users = {
        'aeinstein': {
                'first': 'albert',
                'last': 'einstein',
                'location': 'princeton',
                },
        'mcurie': {
                'first': 'marie',
                'last': 'curie',
                'location': 'paris',
                },
        }

❶ for username, user_info in users.items():
❷         print("\nUsername: " + username)
❸         full_name = user_info['first'] + " " + user_info['last']
          location = user_info['location']
❹         print("\tFull name: " + full_name.title())
          print("\tLocation: " + location.title())
```

At ❸ we start accessing the inner dictionary. The variable `user_info`, which contains the dictionary of user information, has three keys: `'first'`, `'last'`, and `'location'`. We use each key to generate a neatly formatted full name and location for each person, and then print a summary of what we know about each user ❹:

```
Username: aeinstein
        Full name: Albert Einstein
        Location: Princeton
Username: mcurie
        Full name: Marie Curie
        Location: Paris
```

6.5 Summary

In this chapter you learned how to define a dictionary and how to work with the information stored in a dictionary. You learned how to access and modify individual elements in a dictionary, and how to loop through all of the information in a dictionary. You learned to loop through a dictionary's key-value pairs, its keys, and its values. You also learned how to nest multiple dictionaries in a list, nest lists in a dictionary, and nest a dictionary inside a dictionary.

In the next chapter you'll learn about `while` loops and how to accept input from people who are using your programs. This will be an exciting chapter, because you'll learn to make all of your programs interactive: they'll be able to respond to user input.