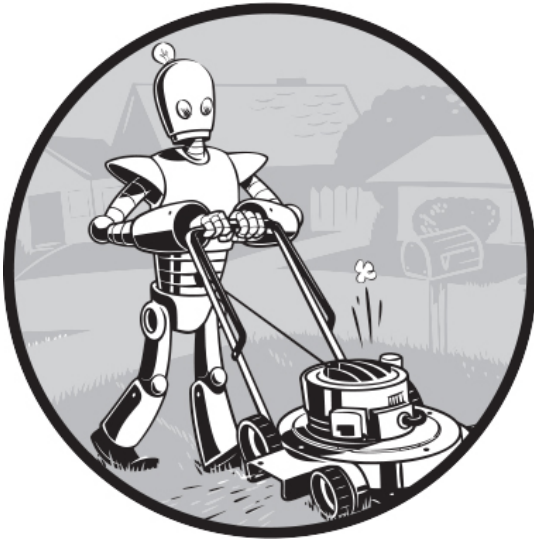


7 DICTIONARIES AND STRUCTURING DATA



This chapter covers the dictionary data type, which provides a flexible way to access and organize data. By combining dictionaries with your knowledge of lists from the previous

chapter, you'll also learn how to create a data structure to model a chessboard.

The Dictionary Data Type

Like a list, a *dictionary* is a mutable collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. These dictionary indexes are called *keys*, and a key with its associated value is called a *key-value pair*.

In code, a dictionary is entered between curly brackets (`{}`). Enter the following into the interactive shell:

```
>>> my_cat = {'size': 'fat', 'color': 'gray', 'age': 17}
```

This assigns a dictionary to the `my_cat` variable. This dictionary's keys are `'size'`, `'color'`, and `'age'`. The values for these keys are `'fat'`, `'gray'`, and `17`, respectively. You can access these values through their keys:

```
>>> my_cat['size']  
'fat'  
>>> 'My cat has ' + my_cat['color'] + ' fur.'  
'My cat has gray fur.'
```

Using dictionaries, you can store multiple pieces of data about the same thing in a single variable. This `my_cat` variable contains three different strings describing my cat, and I can use it as an argument or return value in a function call, saving me from needing to create three separate variables.

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they don't have to start at 0 and can be any number:

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}  
>>> spam[12345]  
'Luggage Combination'  
>>> spam[42]  
'The Answer'  
>>> spam[0]  
Traceback (most recent call last):  
  File "<python-input-0>", line 1, in <module>  
KeyError: 0
```

Dictionaries have keys, not indexes. In this example, while the dictionary in `spam` has integer keys 12345 and 42, it doesn't

have an index 0 through 41 like a list would.

Comparing Dictionaries and Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named `spam` would be `spam[0]`. But there is no "first" item in a dictionary. While the order of items matters for determining whether two lists are the same, you can enter the key-value pairs of a dictionary in any order. Enter the following into the interactive shell:

```
>>> spam = ['cats', 'dogs', 'moose']  
>>> bacon = ['dogs', 'moose', 'cats']  
>>> spam == bacon # The order of list items matters.  
False  
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': 8}  
>>> ham = {'species': 'cat', 'age': 8, 'name': 'Zophie'}  
>>> eggs == ham # The order of dictionary key-value pairs doesn't matter.  
True
```

The `eggs` and `ham` dictionaries are identical values even though we entered their key-value pairs in different orders. Because a dictionary isn't ordered, it isn't a sequence data type and can't be sliced like a list.

Trying to access a key that doesn't exist in a dictionary will result in a `KeyError` error message, much like a list's "out-of-range" `IndexError` error message. Enter the following into

the interactive shell, and notice the error message that shows up because there is no 'color' key:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

Though dictionaries aren't ordered, the fact that you can use arbitrary values as keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values. Open a new file editor window and enter the following code, then save it as *birthdays.py*:

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

while True:
    print('Enter a name: (blank to quit)')
    name = input()
    if name == '':
        break

    ❷ if name in birthdays:
        ❸ print(birthdays[name] + ' is the birthday of ' + name)
    else:
        print('I do not have birthday information for ' + name)
        print('What is their birthday?')
        bday = input()
        ❹ birthdays[name] = bday
        print('Birthday database updated.')
```

The code creates an initial dictionary and stores it in `birthdays` ❶. You can see if the entered name exists as a key in the dictionary with the `in` keyword ❷, just as you did for lists. If the name is in the dictionary, you access the associated value using square brackets ❸; if not, you can add it using the same square bracket syntax combined with the assignment operator ❹.

When you run this program, it will look like this:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Of course, all the data you enter in this program is forgotten when the program terminates. You'll learn how to save data to files on the hard drive in Chapter 10.

Returning Keys and Values

Three dictionary methods will return list-like values of the dictionary's keys, values, or both keys and values: `keys()`, `values()`, and `items()`. The values returned by these methods aren't true lists: they can't be modified and don't have an `append()` method. But these data types (`dict_keys`, `dict_values`, and `dict_items`, respectively) *can* be used in for loops. To see how these methods work, enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
...     print(v)

red
42
```

Here, a for loop iterates over each of the values in the `spam` dictionary. A for loop can also iterate over the keys or both the keys and values:

```
>>> for k in spam.keys():
...     print(k)

color
age
>>> 'color' in spam.keys()
True
>>> 'age' not in spam.keys()
False
>>> 'red' in spam.values()
True
>>> for i in spam.items():
...     print(i)

('color', 'red')
('age', 42)
```

When you use the `keys()`, `values()`, and `items()` methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively, and you can use the `in` and `not in` operators to determine if a value exists as a key or value in the dictionary. Notice that the values in the `dict_items` value returned by the `items()` method are tuples of the key and value.

You can also use the `in` and `not in` operators with the dictionary value itself to check for the existence of a key. This is equivalent to using these operators with the `keys()` method:

```
>>> 'color' in spam
True
>>> 'color' in spam.keys()
True
```

If you want to get an actual list from one of these methods, pass its list-like return value to the `list()` function. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys() # Returns a list-like dict_keys value
dict_keys(['color', 'age'])
>>> list(spam.keys()) # Returns an actual list value
['color', 'age']
```

The `list(spam.keys())` line takes the `dict_keys` value returned from `keys()` and passes it to `list()`, which then returns a list value of `['color', 'age']`.

You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
...     print('Key: ' + str(k) + ' Value: ' + str(v))

Key: color Value: red
Key: age Value: 42
```

This code creates a dictionary with keys `'color'` and `'age'` whose values are `'red'` and `42`, respectively. The for loop iterates over the tuples returned by the `items()` method: `('color', 'red')` and `('age', 42)`. The two variables, `k` and `v`, are assigned the first (the key) and second (the value) values from these tuples. The body of the loop prints out the `k` and `v` variables for each key-value pair.

While you can use many values for keys, you cannot use a list or dictionary as the key in a dictionary. These data types are *unhashable*, which is a concept beyond the scope of this book. If you need a list for a dictionary key, use a tuple instead.

Checking Whether a Key Exists

Checking whether a key exists in a dictionary before accessing that key's value can be tedious. Fortunately, dictionaries have a `get()` method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key doesn't exist.

Enter the following into the interactive shell:

```
>>> picnic_items = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnic_items.get('cups', 0))
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnic_items.get('eggs', 0))
'I am bringing 0 eggs.'
```

Because there is no 'eggs' key in the `picnic_items` dictionary, the `get()` method returns the default value 0. Without using `get()`, the code would have caused an error message, such as in the following example:

```
>>> picnic_items = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnic_items['eggs']) + ' eggs'
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    'I am bringing ' + str(picnic_items['eggs']) + ' eggs'
KeyError: 'eggs'
```

Checking for the existence of a key before accessing the value for that key can prevent your programs from crashing with an error message.

Setting Default Values

You'll often have to set a value in a dictionary for a certain key only if that key doesn't already have a value. Your code might look something like this:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> if 'color' not in spam:
...     spam['color'] = 'black'
...
>>> spam
{'name': 'Pooka', 'age': 5, 'color': 'black'}
```

The `setdefault()` method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key doesn't exist. If the key does exist, the `setdefault()` method returns the key's value. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black') # Sets 'color' to
'black'
>>> spam
{'name': 'Pooka', 'age': 5, 'color': 'black'}
>>> spam.setdefault('color', 'white') # Does nothing
'black'
>>> spam
{'name': 'Pooka', 'age': 5, 'color': 'black'}
```

The first time we call `setdefault()`, the dictionary in `spam` changes to `{'name': 'Pooka', 'age': 5, 'color': 'black'}`. The method returns the value `'black'` because this is now the value set for the key `'color'`. When `spam.setdefault('color', 'white')` is called next, the value for that key is *not* changed to `'white'`, because `spam` already has a key named `'color'`.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string. Open the file editor window and enter the following code, saving it as `characterCount.py`:

```
message = 'It was a bright cold day in April, and the
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

The program loops over each character in the `message` variable's string, counting how often each character appears. The `setdefault()` method call ❶ ensures that the key is in the `count` dictionary (with a default value of 0) so that the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed ❷. When you run this program, the output will look like this:

```
{'I': 1, 't': 6, ' ': 13, 'w': 2, 'a': 4, 's': 3, 'b':
'g': 2, 'h': 3, 'c': 3, 'o': 2, 'l': 3, 'd': 3, 'y': 1,
'p': 1, ',': 1, 'e': 5, 'k': 2, '.': 1}
```

From the output, you can see that the lowercase letter `c` appears three times, the space character appears 13 times, and the uppercase letter `A` appears one time. This program will work no matter what string is inside the `message` variable, even if the string is millions of characters long!

Model Real-World Things Using Data Structures

Even before the internet, it was possible to play a game of chess with someone on the other side of the world. Each player would set up a chessboard at their home, and then they would take turns mailing a postcard to each other describing their move. To do this, the players needed a way to unambiguously describe the state of the board and their moves.

In *algebraic chess notation*, the squares on the chessboard are identified by a number and letter coordinate, as in Figure 7-1.

	a	b	c	d	e	f	g	h	
8	a8	b8	c8	d8	e8	f8		h8	8
7		b7	c7	d7	e7	f7	g7		7
6	a6	b6	c6	d6	e6		g6	h6	6
5	a5	b5	c5	d5	e5	f5	g5	h5	5
4	a4	b4	c4	d4	e4	f4	g4	h4	4
3	a3	b3	c3	d3	e3		g3	h3	3
2	a2	b2	c2	d2	e2	f2	g2		2
1	a1	b1		d1	e1	f1	g1	h1	1
	a	b	c	d	e	f	g	h	

Figure 7-1: The coordinates of a chessboard in algebraic chess notation

The chess pieces use letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move requires specifying the letter of the piece and the coordinates of its destination. A pair of these moves describes what happens in a single turn (with white going first); for instance, the notation “2. *Nf3 Nc6*” indicates that white moved a knight

to f3 and black moved a knight to c6 on the second turn of the game.

There's a bit more to algebraic notation than this, but the point is that you can unambiguously describe a game of chess without needing to be in front of a chessboard. Your opponent can even be on the other side of the world! In fact, you don't even need a physical chess set if you have a good memory: you can just read the mailed chess moves and update boards you have in your imagination.

Computers have good memories. A program on a modern computer can easily store billions of strings, such as '2. Nf3 Nc6'. This is how computers can play chess without having a physical chessboard. They model data to represent a chessboard, and you can write code to work with this model to simulate a chess game.

This is where lists and dictionaries can come in handy. For example, we could come up with our own notation so that the Python dictionary {'h1': 'bK', 'c6': 'wQ', 'g2': 'bB', 'h5': 'bQ', 'e3': 'wK'} could represent the chessboard in Figure 7-2.

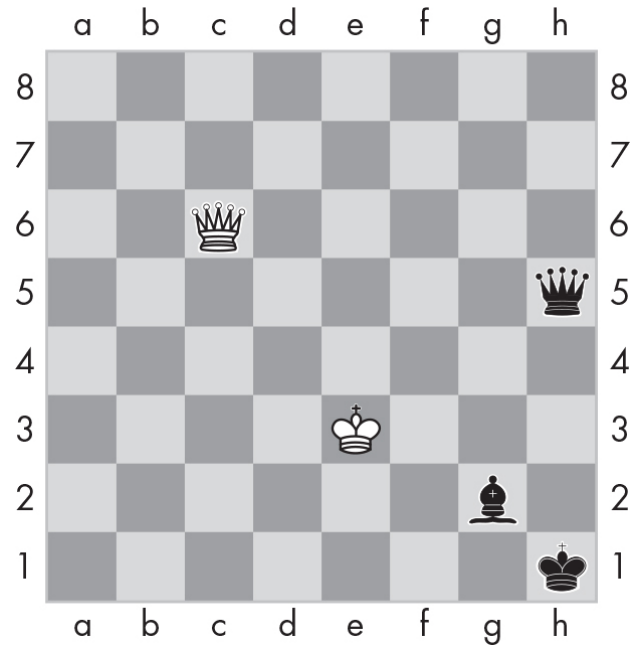


Figure 7-2: A chessboard modeled by the dictionary {'h1': 'bK', 'c6': 'wQ', 'g2': 'bB', 'h5': 'bQ', 'e3': 'wK'}

Let's use this data structure scheme to create our own interactive chessboard program.

Project 1: Interactive Chessboard Simulator

Even the earliest computers performed calculations far faster than any human, but back then, people considered chess a true demonstration of computational intelligence. We won't create our own chess-playing program here. (That would require its own book!) But we can create an interactive chessboard program with what we've discussed so far.

You don't need to know the rules of chess for this program. Just know that chess is played on an 8×8 board with white and black pieces called pawns, knights, bishops, rooks, queens, and kings. The upper-left and lower-right squares of the board should be white, and our program assumes the background of the output window is black (unlike the white background of a paper book). Our chessboard program is just a board with pieces on it; it doesn't even enforce the rules for how pieces move. We'll use text characters to "draw" a chessboard, such as the one shown in Figure 7-3.

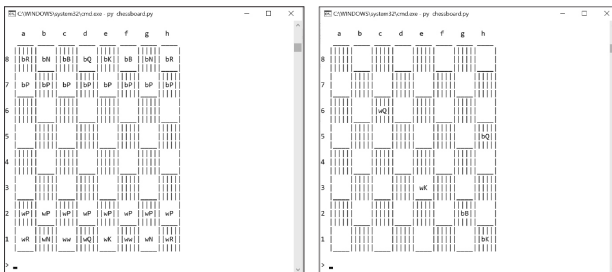


Figure 7-3: The non-graphical, text-based output of the chessboard program

Graphics would be nice and would make the pieces more readily identifiable, but we've captured all the information about the pieces without them. This text-based approach allows us to write the program with just the `print()` function and doesn't require us to install any sort of graphics library like Pygame (discussed in my book *Invent Your Own Computer Games with Python* [No Starch Press, 2016]) for our program.

First, we need to design a data structure that can represent a chessboard and any possible configuration of pieces on it. The example from the previous section works: the board is a Python dictionary with string keys 'a1' to 'h8' to represent squares on the board. Note that these strings are always two characters long. Also, the letter is always lowercase and comes before the number. This specificity is important; we'll use these details in the code.

To represent the pieces, we'll also use two-character strings, where the first letter is a lowercase 'w' or 'b' to indicate the white or black color, and the second letter is an uppercase 'P', 'N', 'B', 'R', 'Q', or 'K' to represent the kind of piece. Figure 7-4 shows each piece, along with its string representation.

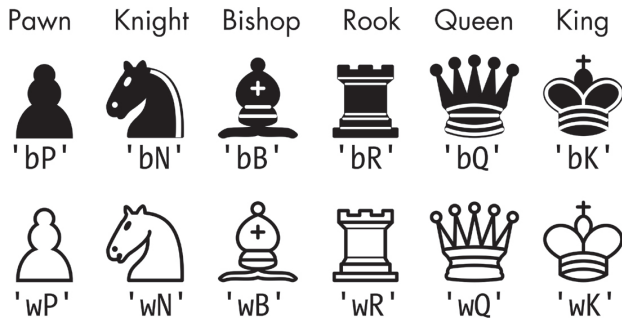


Figure 7-4: The two-character string representations for each chess piece

The keys of the Python dictionary identify the squares of the board and the values identify the piece on that square. The absence of a key in the dictionary represents an empty square. A dictionary works well for storing this information: keys in a dictionary can be used only once, and squares on a chessboard can have only one piece on them at a time.

Step 1: Set Up the Program

The first part of the program imports the `sys` module for its `exit()` function and the `copy` module for its `copy()` function. At the start of the game, the white and black players have 16 pieces each. The `STARTING_PIECES` constant will hold a chessboard dictionary with all the proper pieces in their correct starting positions:

```
import sys, copy

STARTING_PIECES = {'a8': 'bR', 'b8': 'bN', 'c8': 'bB',
                   'e8': 'bK', 'f8': 'bB', 'g8': 'bN', 'h8': 'bR', 'a7':
                   'c7': 'bP', 'd7': 'bP', 'e7': 'bP', 'f7': 'bP', 'g7':
                   'a1': 'wR', 'b1': 'wN', 'c1': 'wB', 'd1': 'wQ', 'e1':
                   'g1': 'wN', 'h1': 'wR', 'a2': 'wP', 'b2': 'wP', 'c2':
                   'e2': 'wP', 'f2': 'wP', 'g2': 'wP', 'h2': 'wP'}
```

(This code is a bit hard to type. You can copy and paste it from <https://autbor.com/3/chessboard.py> (<https://autbor.com/3/chessboard.py>).) Whenever the program needs to reset the chessboard to the starting setup, it can copy `STARTING_PIECES` with the `copy.copy()` function.

Step 2: Create a Chessboard Template

The `BOARD_TEMPLATE` variable will contain a string that acts as a template for a chessboard. The program can insert the strings of individual pieces into it before printing. By using three double-quote characters in a row, we can create a *multiline string* that spans several lines of code. The multiline string ends with another three double-quote characters. This Python syntax is easier than trying to fit everything on a single line with `\n` escape characters. You'll learn more about multiline strings in the next chapter.

```

BOARD_TEMPLATE = """
    a      b      c      d      e      f      g      h
 8 | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| |
   | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| |
 7 | {} ||||| {} ||||| {} ||||| {} ||||| {} ||||| {} |||||
   | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| |
 6 | ||||| {} ||||| {} ||||| {} ||||| {} ||||| {} |||||
   | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| |
 5 | {} ||||| {} ||||| {} ||||| {} ||||| {} ||||| {} |||||
   | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| |
 4 | ||||| {} ||||| {} ||||| {} ||||| {} ||||| {} |||||
   | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| |
 3 | {} ||||| {} ||||| {} ||||| {} ||||| {} ||||| {} |||||
   | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| |
 2 | ||||| {} ||||| {} ||||| {} ||||| {} ||||| {} |||||
   | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| |
 1 | {} ||||| {} ||||| {} ||||| {} ||||| {} ||||| {} |||||
   | ||||| | ||||| | ||||| | ||||| | ||||| | ||||| |
"""
WHITE_SQUARE = '|||'
BLACK_SQUARE = ' {} '

```

The pairs of curly brackets represent places in the string where we'll insert chess piece strings such as 'wR' or 'bQ'. If the square is empty, the program will insert the WHITE_SQUARE or BLACK_SQUARE string instead, which I'll explain in more detail when we discuss the print_chessboard() function.

Step 3: Print the Current Chessboard

We'll define a print_chessboard() function that accepts the chessboard dictionary, then prints a chessboard on the screen that reflects the pieces on this board. We'll call the format() string method on the BOARD_TEMPLATE string, passing the method a list of strings. The format() method returns a new string with the {} pairs in BOARD_TEMPLATE replaced by the strings in the passed-in list. You'll learn more about format() in the next chapter.

Let's take a look at the code in print_chessboard():

```

def print_chessboard(board):
    squares = []
    is_white_square = True
    for y in '87654321':
        for x in 'abcdefgh':
            #print(x, y, is_white_square) # DEBUG: Show

```

There are 64 squares on a chessboard and 64 {} pairs in the BOARD_TEMPLATE string. We must build up a list of 64 strings to replace these {} pairs. We store this list in the squares variable. The strings in this list represent either chess pieces, like 'wB' and 'bQ', or empty squares. Depending on whether the empty square is white or black, we must use the WHITE_SQUARE string ('| |') or the BLACK_SQUARE string (' '). We'll use a Boolean value in the is_white_square variable to keep track of which squares are white and which are black.

Two nested for loops will loop over all 64 squares on the board, starting with the upper-left square and going right to left, then top to bottom. The square in the upper left is a white square, so we'll start is_white_square as True. Remember that for loops can loop over the integers given by range(), the value in a list, or the individual characters in a string. In these two for loops, the y and x variables take on the characters in the strings '87654321' and 'abcdefgh', respectively. To see the order in which the code loops over the squares (along with the color of each square), uncomment the print(x, y, is_white_square) line of code before running the program.

The code inside the for loops builds up the squares list with the appropriate strings:

```
if x + y in board.keys():
    squares.append(board[x + y])
else:
    if is_white_square:
        squares.append(WHITE_SQUARE)
    else:
        squares.append(BLACK_SQUARE)
    is_white_square = not is_white_square
    is_white_square = not is_white_square

print(BOARD_TEMPLATE.format(*squares))
```

The strings in the x and y loop variables concatenate together to form a two-character square string. For example, if x is 'a' and y is '8', then x + y evaluates to 'a8', and x + y in board.keys() checks if this string exists as a key in the chessboard dictionary. If it does, the code appends the chess piece string for that square to the end of the squares list.

If it does not, the code must append the blank square string in WHITE_SQUARE or BLACK_SQUARE, depending on the value in is_white_square. Once the code is done processing this chessboard square, it toggles the Boolean value in is_white_square to its opposite value (because the next square over will be the opposite color). The variable needs to be toggled again after finishing a row at the end of the outermost for loop.

After the loops have finished, the `squares` list contains 64 strings. However, the `format()` string method doesn't take a single list argument, but rather one string argument per `{}` pair to replace. The asterisk `*` next to the `squares` tells Python to pass the values in this list as individual arguments. This is a bit subtle, but imagine that you have a list `spam = ['cat', 'dog', 'rat']`. If you call `print(spam)`, Python will print the list value, along with its square brackets, quotes, and commas. However, calling `print(*spam)` is equivalent to calling `print('cat', 'dog', 'rat')`, which simply prints `cat dog rat`. I call this *star syntax*.

The `print_chessboard()` function is written to work with the specific data structure we use to represent chessboards: a Python dictionary with keys of square strings, like `'a8'`, and values of pieces of strings, like `'bQ'`. If we had designed our data structure differently, we'd have to write our function differently too. The `print_chessboard()` prints out a text-based representation of the board, but if we were using a graphics library like Pygame to render the chessboard, we could still use this Python dictionary to represent the chessboard configuration.

Step 4: Manipulate the Chessboard

Now that we have a way to represent a chessboard in a Python dictionary and a function to display a chessboard based on that dictionary, let's write code that moves pieces on the board by manipulating the keys and values of the

dictionary. After the `print_chessboard()` function's `def` block, the main part of the program displays text explaining how to use the interactive chessboard program:

```
print('Interactive Chessboard')
print('by Al Sweigart al@inventwithpython.com')
print()
print('Pieces:')
print('  w - White, b - Black')
print('  P - Pawn, N - Knight, B - Bishop, R - Rook, Q - Queen')
print('Commands:')
print('  move e2 e4 - Moves the piece at e2 to e4')
print('  remove e2 - Removes the piece at e2')
print('  set e2 wP - Sets square e2 to a white pawn')
print('  reset - Resets pieces back to their starting positions')
print('  clear - Clears the entire board')
print('  fill wP - Fills entire board with white pawns')
print('  quit - Quits the program')
```

The program can move pieces, remove pieces, set squares with a piece, reset the board, and clear the board by changing the chessboard dictionary:

```
main_board = copy.copy(STARTING_PIECES)
while True:
    print_chessboard(main_board)
    response = input('> ').split()
```

To begin, the `main_board` variable receives a copy of the `STARTING_PIECES` dictionary, which is a dictionary of all chess pieces in their standard starting positions. The execution enters an infinite loop that allows the user to enter commands. For example, if the user enters **move e2 e4** after `input()` is called, the `split()` method returns the list `['move', 'e2', 'e4']`, which the program then stores in the `response` variable. The first item in the `response` list, `response[0]`, will be the command the user wants to carry out:

```
if response[0] == 'move':  
    main_board[response[2]] = main_board[response[1]]  
    del main_board[response[1]]
```

If the user enters something like **move e2 e4**, then `response[0]` is `'move'`. We can “move” a piece from one square to another by first copying the piece in `main_board` from the old square (in `response[1]`) to the new square (in `response[2]`). Then, we can delete the key-value pair for the old square in `main_board`. This has the effect of making it seem like the piece has moved (though we won’t see this change until we call `print_chessboard()` again).

Our interactive chessboard simulator doesn’t check if this is a valid move to make. It just carries out the commands given

by the user. If the user enters something like **remove e2**, the program will set `response` to `['remove', 'e2']`:

```
elif response[0] == 'remove':  
    del main_board[response[1]]
```

By deleting the key-value pair at key `response[1]` from `main_board`, we make the piece disappear from the board. If the user enters something like **set e2 wP** to add a white pawn to e2, the program will set `response` to `['set', 'e2', 'wP']`:

```
elif response[0] == 'set':  
    main_board[response[1]] = response[2]
```

We can create a new key-value pair with the key `response[1]` and value `response[2]` in `main_board` to add this piece to the board. If the user enters `reset`, `response` is simply `['reset']`, and we can set the board to its starting configuration by copying the `STARTING_PIECES` dictionary to `main_board`:

```
elif response[0] == 'reset':  
    main_board = copy.copy(STARTING_PIECES)
```

If the user enters `clear`, response is simply `['clear']`, and we can remove all pieces from the board by setting `main_board` to an empty dictionary:

```
elif response[0] == 'clear':  
    main_board = {}
```

If the user enters `fill wP`, response is `['fill', 'wP']`, and we change all 64 squares to the string `'wP'`:

```
elif response[0] == 'fill':  
    for y in '87654321':  
        for x in 'abcdefgh':  
            main_board[x + y] = response[1]
```

The nested for loops will loop over every square, setting the `x + y` key to `response[1]`. There's no real reason to put 64 white pawns on a chessboard, but this command demonstrates how easy it is to manipulate the chessboard data structure however we want. Finally, the user can quit the program by entering `quit`:

```
elif response[0] == 'quit':  
    sys.exit()
```

After carrying out the command and modifying `main_board`, the execution jumps back to the start of the `while` loop to display the changed board and accept a new command from the user.

This interactive chessboard program doesn't restrict what pieces you can place or move. It simply uses a dictionary as a representation of pieces on a chessboard and has one function for displaying such dictionaries on the screen in a way that looks like a chessboard. We can model all real-world objects or processes by designing data structures and writing functions to work with those data structures. If you'd like to see another example of modeling a game board with data structures, my other book, *The Big Book of Small Python Projects* (No Starch Press, 2021), has a working tic-tac-toe program.

Nested Dictionaries and Lists

As you model more complicated things, you may find you need dictionaries and lists that contain other dictionaries and lists. Lists are useful for holding an ordered series of values, and dictionaries are useful for associating keys with values. For example, here's a program that uses a dictionary to contain dictionaries of items guests are bringing to a picnic. The `total_brought()`

function can read this data structure and calculate the total number of each item type. Enter the following code in a new program saved as *guestpicnic.py*:

```
all_guests = {'Alice': {'apples': 5, 'pretzels': 12},
              'Bob': {'ham sandwiches': 3, 'apples': 2},
              'Carol': {'cups': 3, 'apple pies': 1}}

def total_brought(guests, item):
    num_brought = 0
    ❶ for k, v in guests.items():
        ❷ num_brought = num_brought + v.get(item, 0)
    return num_brought

print('Number of things being brought:')
print(' - Apples      ' + str(total_brought(all_guests, 'apples')))
print(' - Cups        ' + str(total_brought(all_guests, 'cups')))
print(' - Cakes        ' + str(total_brought(all_guests, 'cakes')))
print(' - Ham Sandwiches ' + str(total_brought(all_guests, 'ham sandwiches')))
print(' - Apple Pies   ' + str(total_brought(all_guests, 'apple pies')))
```

Inside the `total_brought()` function, the `for` loop iterates over the key-value pairs in `guests` ❶. Inside the loop, the string of the guest's name is assigned to `k`, and the dictionary of picnic items they're bringing is assigned to `v`. If the item parameter exists as a key in this dictionary, its value (the quantity) is added to `num_brought` ❷. If it doesn't exist as a

key, the `get()` method returns `0` to be added to `num_brought`.

The output of this program looks like this:

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies 1
```

The number of items brought to a picnic may seem like such a simple thing to model that you wouldn't need to bother with writing a program to do it. But realize that this same `total_brought()` function could easily handle a dictionary that contains thousands of guests, each bringing thousands of different picnic items. In that case, having this information in a data structure, along with the `total_brought()` function, would save you a lot of time!

You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly. When you first begin programming, don't worry so much about the "right" way to model data. As you gain more experience, you may come up with more efficient models; the important thing is that the data model works for your program's needs.

Summary

You learned all about dictionaries in this chapter. Lists and dictionaries are values that can contain multiple values, including other lists and dictionaries. Dictionaries are useful because you can map one item (the key) to another item (the value), whereas lists simply contain a series of values in order. Code can access values inside a dictionary using square brackets just as with lists. Instead of integer indexes, dictionaries can have keys of a variety of data types: integers, floats, strings, or tuples. By organizing a program's values into data structures, you can create representations of real-world objects, such as the chessboard modeled in this chapter.

Practice Questions

1. What does the code for an empty dictionary look like?
2. What does a dictionary value with a key 'foo' and a value 42 look like?

3. What is the main difference between a dictionary and a list?
4. What happens if you try to access `spam['foo']` if `spam` is `{'bar': 100}`?
5. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.keys()`?
6. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.values()`?
7. What is a shortcut for the following code?

```
if 'color' not in spam:  
    spam['color'] = 'black'
```

8. What module and function can be used to “pretty-print” dictionary values?

Practice Programs

For practice, write programs to do the following tasks.

Chess Dictionary Validator

In this chapter, we used the dictionary value `{'h1': 'bK', 'c6': 'wQ', 'g2': 'bB', 'h5': 'bQ', 'e3': 'wK'}` to represent a chessboard. Write a function named `isValidChessBoard()` that takes a dictionary argument and returns `True` or `False` depending on whether the board is valid.

A valid board will have exactly one black king and exactly one white king. Each player can have at most 16 pieces, of which only eight can be pawns, and all pieces must be on a valid square from `'1a'` to `'8h'`. That is, a piece can't be on square `'9z'`. The piece names should begin with either a `'w'` or a `'b'` to represent white or black, followed by `'pawn'`, `'knight'`, `'bishop'`, `'rook'`, `'queen'`, or `'king'`. This function should detect when a bug has resulted in an improper chessboard. (This isn't an exhaustive list of requirements, but it is close enough for this exercise.)

Fantasy Game Inventory

Say you're creating a medieval fantasy video game. The data structure to model the player's inventory is a dictionary whose keys are strings describing the item in the inventory and whose values are integers detailing how many of that item the player has. For example, the dictionary value `{'rope': 1, 'torch': 6, 'gold coin': 42,`

`'dagger': 1, 'arrow': 12}` means the player has one rope, six torches, 42 gold coins, and so on.

Write a function named `display_inventory()` that would take any possible "inventory" and display it like the following:

```
Inventory:
12 arrow
42 gold coin
1 rope
6 torch
1 dagger
Total number of items: 62
```

Hint: You can use a `for` loop to loop through all keys in a dictionary.

```
stuff = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}

def display_inventory(inventory):
    print("Inventory:")
    item_total = 0
    for k, v in inventory.items():
        # FILL THIS PART IN
    print("Total number of items: " + str(item_total))

display_inventory(stuff)
```

List-to-Dictionary Loot Conversion

Imagine that the same fantasy video game represents a vanquished dragon's loot as a list of strings, like this:

```
dragon_loot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
```

Write a function named `add_to_inventory(inventory, added_items)`. The `inventory` parameter is a dictionary representing the player's inventory (as in the previous project) and the `added_items` parameter is a list, like `dragon_loot`. The `add_to_inventory()` function should return a dictionary that represents the player's updated inventory. Note that the `added_items` list can contain multiples of the same item. Your code could look something like this:

```
def add_to_inventory(inventory, added_items):
    # Your code goes here.

inv = {'gold coin': 42, 'rope': 1}
dragon_loot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
inv = add_to_inventory(inv, dragon_loot)
display_inventory(inv)
```

The previous program (with your `display_inventory()` function from the previous project) would output the following:

```
Inventory:
45 gold coin
1 rope
1 ruby
1 dagger
```

```
Total number of items: 48
```