We set the variable `active` to `True` ❶ so the program starts in an active state. Doing so makes the `while` statement simpler because no comparison is made in the `while` statement itself; the logic is taken care of in other parts of the program. As long as the `active` variable remains `True`, the loop will continue running ❷.

In the `if` statement inside the `while` loop, we check the value of `message` once the user enters their input. If the user enters `'quit'` ❸, we set `active` to `False`, and the `while` loop stops. If the user enters anything other than `'quit'` ❹, we print their input as a message.

This program has the same output as the previous example where we placed the conditional test directly in the `while` statement. But now that we have a flag to indicate whether the overall program is in an active state, it would be easy to add more tests (such as `elif` statements) for events that should cause `active` to become `False`. This is useful in complicated programs like games in which there may be many events that should each make the program stop running. When any of these events causes the active flag to become `False`, the main game loop will exit, a *Game Over* message can be displayed, and the player can be given the option to play again.

Using break to Exit a Loop

To exit a `while` loop immediately without running any remaining code in the loop, regardless of the results of any conditional test, use the `break` statement. The `break` statement directs the flow of your program; you can use it to control which lines of code are executed and which aren't, so the program only executes code that you want it to, when you want it to.

For example, consider a program that asks the user about places they've visited. We can stop the `while` loop in this program by calling `break` as soon as the user enters the `'quit'` value:

cities.py

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "

❶ while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print(f"I'd love to go to {city.title()}!")
```

A loop that starts with `while True` ❶ will run forever unless it reaches a `break` statement. The loop in this program continues asking the user to enter the names of cities they've been to until they enter `'quit'`. When they enter `'quit'`, the `break` statement runs, causing Python to exit the loop:

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) New York
I'd love to go to New York!
```

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) San Francisco
I'd love to go to San Francisco!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

NOTE *You can use the* break *statement in any of Python's loops. For example, you could use* break *to quit a* for *loop that's working through a list or a dictionary.*

Using continue in a Loop

Rather than breaking out of a loop entirely without executing the rest of its code, you can use the `continue` statement to return to the beginning of the loop based on the result of a conditional test. For example, consider a loop that counts from 1 to 10 but prints only the odd numbers in that range:

counting.py

```
current_number = 0
while current_number < 10:
❶    current_number += 1
    if current_number % 2 == 0:
        continue

    print(current_number)
```

First we set `current_number` to 0. Because it's less than 10, Python enters the `while` loop. Once inside the loop, we increment the count by 1 at ❶, so `current_number` is 1. The `if` statement then checks the modulo of `current_number` and 2. If the modulo is 0 (which means `current_number` is divisible by 2), the `continue` statement tells Python to ignore the rest of the loop and return to the beginning. If the current number is not divisible by 2, the rest of the loop is executed and Python prints the current number:

```
1
3
5
7
9
```

Avoiding Infinite Loops

Every `while` loop needs a way to stop running so it won't continue to run forever. For example, this counting loop should count from 1 to 5:

counting.py

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

But if you accidentally omit the line `x += 1` (as shown next), the loop will run forever:

```
# This loop runs forever!
x = 1
while x <= 5:
    print(x)
```

Now the value of `x` will start at `1` but never change. As a result, the conditional test `x <= 5` will always evaluate to `True` and the `while` loop will run forever, printing a series of 1s, like this:

```
1
1
1
1
--snip--
```

Every programmer accidentally writes an infinite `while` loop from time to time, especially when a program's loops have subtle exit conditions. If your program gets stuck in an infinite loop, press CTRL-C or just close the terminal window displaying your program's output.

To avoid writing infinite loops, test every `while` loop and make sure the loop stops when you expect it to. If you want your program to end when the user enters a certain input value, run the program and enter that value. If the program doesn't end, scrutinize the way your program handles the value that should cause the loop to exit. Make sure at least one part of the program can make the loop's condition `False` or cause it to reach a `break` statement.

NOTE *Sublime Text and some other editors have an embedded output window. This can make it difficult to stop an infinite loop, and you might have to close the editor to end the loop. Try clicking in the output area of the editor before pressing CTRL-C, and you should be able to cancel an infinite loop.*

TRY IT YOURSELF

7-4. Pizza Toppings: Write a loop that prompts the user to enter a series of pizza toppings until they enter a `'quit'` value. As they enter each topping, print a message saying you'll add that topping to their pizza.

7-5. Movie Tickets: A movie theater charges different ticket prices depending on a person's age. If a person is under the age of 3, the ticket is free; if they are between 3 and 12, the ticket is $10; and if they are over age 12, the ticket is $15. Write a loop in which you ask users their age, and then tell them the cost of their movie ticket.

(continued)

7-6. Three Exits: Write different versions of either Exercise 7-4 or Exercise 7-5 that do each of the following at least once:

- Use a conditional test in the `while` statement to stop the loop.
- Use an `active` variable to control how long the loop runs.
- Use a `break` statement to exit the loop when the user enters a `'quit'` value.

7-7. Infinity: Write a loop that never ends, and run it. (To end the loop, press CTRL-C or close the window displaying the output.)

Using a while Loop with Lists and Dictionaries

So far, we've worked with only one piece of user information at a time. We received the user's input and then printed the input or a response to it. The next time through the `while` loop, we'd receive another input value and respond to that. But to keep track of many users and pieces of information, we'll need to use lists and dictionaries with our `while` loops.

A `for` loop is effective for looping through a list, but you shouldn't modify a list inside a `for` loop because Python will have trouble keeping track of the items in the list. To modify a list as you work through it, use a `while` loop. Using `while` loops with lists and dictionaries allows you to collect, store, and organize lots of input to examine and report on later.

Moving Items from One List to Another

Consider a list of newly registered but unverified users of a website. After we verify these users, how can we move them to a separate list of confirmed users? One way would be to use a `while` loop to pull users from the list of unconfirmed users as we verify them and then add them to a separate list of confirmed users. Here's what that code might look like:

confirmed_users.py

```
# Start with users that need to be verified,
#  and an empty list to hold confirmed users.
❶ unconfirmed_users = ['alice', 'brian', 'candace']
confirmed_users = []

# Verify each user until there are no more unconfirmed users.
#  Move each verified user into the list of confirmed users.
❷ while unconfirmed_users:
❸     current_user = unconfirmed_users.pop()

      print(f"Verifying user: {current_user.title()}")
❹     confirmed_users.append(current_user)
```

at least once. Once inside the loop, Python removes the first instance of `'cat'`, returns to the `while` line, and then reenters the loop when it finds that `'cat'` is still in the list. It removes each instance of `'cat'` until the value is no longer in the list, at which point Python exits the loop and prints the list again:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Filling a Dictionary with User Input

You can prompt for as much input as you need in each pass through a `while` loop. Let's make a polling program in which each pass through the loop prompts for the participant's name and response. We'll store the data we gather in a dictionary, because we want to connect each response with a particular user:

mountain_poll.py

```
responses = {}

# Set a flag to indicate that polling is active.
polling_active = True

while polling_active:
    # Prompt for the person's name and response.
❶   name = input("\nWhat is your name? ")
    response = input("Which mountain would you like to climb someday? ")

    # Store the response in the dictionary.
❷   responses[name] = response

    # Find out if anyone else is going to take the poll.
❸   repeat = input("Would you like to let another person respond? (yes/ no) ")
    if repeat == 'no':
        polling_active = False

# Polling is complete. Show the results.
print("\n--- Poll Results ---")
❹ for name, response in responses.items():
    print(f"{name} would like to climb {response}.")
```

The program first defines an empty dictionary (`responses`) and sets a flag (`polling_active`) to indicate that polling is active. As long as `polling_active` is `True`, Python will run the code in the `while` loop.

Within the loop, the user is prompted to enter their name and a mountain they'd like to climb ❶. That information is stored in the `responses` dictionary ❷, and the user is asked whether or not to keep the poll running ❸. If they enter `yes`, the program enters the `while` loop again. If they enter `no`, the `polling_active` flag is set to `False`, the `while` loop stops running, and the final code block at ❹ displays the results of the poll.

If you run this program and enter sample responses, you should see output like this:

```
What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes

What is your name? Lynn
Which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/ no) no

--- Poll Results ---
Lynn would like to climb Devil's Thumb.
Eric would like to climb Denali.
```

TRY IT YOURSELF

7-8. Deli: Make a list called `sandwich_orders` and fill it with the names of various sandwiches. Then make an empty list called `finished_sandwiches`. Loop through the list of sandwich orders and print a message for each order, such as `I made your tuna sandwich.` As each sandwich is made, move it to the list of finished sandwiches. After all the sandwiches have been made, print a message listing each sandwich that was made.

7-9. No Pastrami: Using the list `sandwich_orders` from Exercise 7-8, make sure the sandwich `'pastrami'` appears in the list at least three times. Add code near the beginning of your program to print a message saying the deli has run out of pastrami, and then use a `while` loop to remove all occurrences of `'pastrami'` from `sandwich_orders`. Make sure no pastrami sandwiches end up in `finished_sandwiches`.

7-10. Dream Vacation: Write a program that polls users about their dream vacation. Write a prompt similar to *If you could visit one place in the world, where would you go?* Include a block of code that prints the results of the poll.

Summary

In this chapter you learned how to use `input()` to allow users to provide their own information in your programs. You learned to work with both text and numerical input and how to use `while` loops to make your programs run as long as your users want them to. You saw several ways to control the flow of a `while` loop by setting an `active` flag, using the `break` statement, and

using the `continue` statement. You learned how to use a `while` loop to move items from one list to another and how to remove all instances of a value from a list. You also learned how `while` loops can be used with dictionaries.

In Chapter 8 you'll learn about *functions*. Functions allow you to break your programs into small parts, each of which does one specific job. You can call a function as many times as you want, and you can store your functions in separate files. By using functions, you'll be able to write more efficient code that's easier to troubleshoot and maintain and that can be reused in many different programs.