7

DANCE PARTIES AND FLOWER PARADES WITH WHILE LOOPS

Loops make it easy to repeat code again and again. Instead of copying and pasting the same code, you can use a loop to repeat the code as many times as you want. You'll use loops in this chapter to make your programs repeat without having to rerun them. We'll focus on one type of Python loop known as the `while` loop.

A SIMPLE WHILE LOOP

You use `while` loops to repeat blocks of code. Similar to `if` statements, a `while` loop will execute the code inside it as long as a condition is `True`. That is, a condition must be met in order for the body of the statement to run.

The difference between a `while` loop and an `if` statement is that the code in the `if` statement executes only once at the most, whereas the code in the `while` loop can repeat many times. Programmers call the repeating of code *iteration*. When a loop repeats, you say it *iterates*.

For example, this code uses a `while` loop to print the numbers 1 to 5:

```
count = 1
while count <= 5:
    print(count)
    count += 1
print("Loop finished")
```

The `count` variable records the number of times that the loop has repeated. It starts with the value of 1. The condition in the `while` loop checks whether the count is less than or equal to 5.

NOTE *In Chapter 3 you learned that `+=` is a shorthand operator. You could use the standard addition operator `count = count + 1` to do the same thing.*

The first time the loop runs, the value of `count` is `1`, which is less than 5. The condition of the loop is `True`, and the body of the loop runs. Next, the program prints the value of `count` to the Python shell, and then it adds 1 to the value of `count`. The `while` loop now starts again and checks the condition again, going through each step until the `count` variable is greater than 5.

Outside the loop is one final line, which prints `"Loop finished"`.

Save this program and run it; you should see the following output:

```
1
2
3
4
5
Loop finished
```

Try experimenting a bit with the code. Change the conditions so you list more than 5 numbers or change the amount by which the `count` variable increases. Here's a refresher on how the code works. The `while` statement follows these steps:

1. Check whether the condition is `True`.
2. If the condition is `True`:
   a. Execute the body of code.
   b. Repeat step 1.
3. If the condition is `False`:
   a. Ignore the body of code.
4. Continue to the line after the `while` loop block.

Let's try using a `while` loop in Minecraft to teleport to lots of new places!

MISSION #33: A RANDOM TELEPORTATION TOUR

In Mission #3 (page 40), you teleported the player to different positions in the game. Let's rewrite that program using a `while` loop so you can repeat the teleportation again and again.

By looping some code that will teleport the player to a random location, you can make the program more powerful *and* a lot easier to read. Cool, huh?

The following code will teleport the player to a random location once by picking random values in the game world for the variables `x`, `y`, and `z`. Then it will set the player's position using those variables.

```
import random
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ # Add the count variable here
❷ # Start the while loop here
❸ x = random.randint(-127, 127)  # Indent the code from this line
  y = random.randint(0, 64)
  z = random.randint(-127, 127)

  mc.player.setTilePos(x, y, z)
❹ # Add 1 to the value of the count variable here
```

Right now, however, the code will only teleport the player once. Although that's pretty cool, you can make it totally awesome. Let's write a loop so the code repeats five times, making this quite a whirlwind tour.

To change the code to use a loop, follow these four steps:

1. Create a `count` variable to control the loop ❶.
2. Add a `while` loop with a condition based on `count` ❷.
3. Indent the body of the `while` statement ❸.
4. Increment the value of `count` with each loop ❹.

The purpose of the `count` variable and the `count` increment is to keep track of the number of times the loop has repeated. I'll talk more about them in the next section. For now, all you need to know is that `count` lets us control how many times this code repeats.

Listing 7-1 shows the code with the changes added.

random Teleport.py

```
import random
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

count = 0
while count < 5:
    x = random.randint(-127, 127)
    y = random.randint(0, 64)
    z = random.randint(-127, 127)
```

```
    mc.player.setTilePos(x, y, z)
    count += 1
```

Listing 7-1: Code to randomly teleport the player around the game world

Copy Listing 7-1 into a new file, save it as *randomTeleport.py* in a new folder called *whileLoops*, and run the code. You should see the player zip around the Minecraft world. But the code runs far too quickly! The entire journey is over in less than a second. Let's fix that together.

You'll use the `time` module to slow down the code. Follow these steps:

1. On the first line of the program, add the statement `import time`. This imports Python's `time` module, which contains a set of handy functions related to timing and more.
2. Add the line `time.sleep(10)` at the end of the body of your `while` loop to add a delay of 10 seconds to your program. Make sure you indent this new final line of your program so it's within the `while` loop!

Save the program and run it. Now the player should teleport to a new random location every 10 seconds. Figure 7-1 shows my program running.

Figure 7-1: Every 10 seconds, the program teleports me to a new location.

BONUS OBJECTIVE: SLEEP TIGHT

At the moment, the program will wait for 10 seconds at the end of every loop. What happens if you move the `time.sleep(10)` statement to the start of the loop?

CONTROLLING LOOPS WITH A COUNT VARIABLE

Count variables are a common way of storing the number of times a program has repeated. You've seen these variables in action a few times now. Let's look at another example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

The `while` loop's condition tests that the value of the `count` variable is less than 5. In the body of the loop, I've changed the value of the `count` variable to record the number of times the count has repeated. Adding to the value of a `count` variable is called *incrementing*.

The last line of this code increases the value of the `count` variable by 1. Each time the code repeats, it will check the new value of the `count` variable to see whether it is less than 5. When it is equal to or greater than 5, the loop will stop.

If you forget to increment the `count` variable, you'll end up with an *infinite loop*, which will repeat the loop forever, as shown in the following example:

```
count = 0
while count < 5:
    print(count)
```

The value of `count` is always 0 because it's never incremented. So, the condition of the loop will always be `True`, and the loop will repeat *forever*. If you don't believe me, try running the code!

```
0
0
0
0
0
--snip--
```

To break the execution of this infinite program, press CTRL-C. To correct the code, just add the line `count += 1` to the loop's body. Now you won't get trapped in an infinite loop. Phew!

Counts don't always have to be incremented by 1. In some situations you may want to increment the count by a different value. In the following example, the count is incremented by 2 every time; the result is that the code prints all the even numbers between 0 and 100:

```
count = 0
while count < 100:
    print(count)
    count += 2
```

You can also count backward using a negative number to *decrement* the value of the count. The following code counts *down* from 100 to 1:

```
count = 100
while count > 0:
    print(count)
    count -= 1
```

The only difference between this example and the previous examples is the condition. Here I've used a greater than comparator (`>`). As long as the count is greater than 0, the loop continues; when the count reaches 0, the loop stops.

NOTE *The variable used to control a loop isn't always called* `count`*. You could call it* `repeats` *or anything else you want. If you look at other people's code, you will see a huge range of different names.*

MISSION #34: THE WATERY CURSE

Let's try something a bit nasty and write a curse for the player that lasts for just a short time. Curses in video games might *debuff* the character in some way, such as slowing them down or making them weaker, often for just a little while.

We'll create a curse program that places a flowing water block at the player's position once a second for 30 seconds. This will make it difficult for the player to move without being pushed around by flowing water.

The following code places a flowing water block at the player's position:

waterCurse.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getPos()
mc.setBlock(pos.x, pos.y, pos.z, 8)
```

This code will place a water block at the player's current position only once. It is your task to make it repeat. The final code should repeat 30 times, and each iteration of the loop should last 1 second.

Save this code as *waterCurse.py* in the *whileLoops* folder and run it once to make sure it works. You should see a single water block appear at the player's position before the program stops.

Let's talk through what to add next to make this curse last. Use what you learned about `while` loops and `count` variables to do the following:

1. Add a `count` variable to the program.
2. Add a loop to the program to repeat the last two lines of code. The loop should repeat 30 times.
3. Increment the `count` variable at the end of the loop.
4. Import the `time` module (on the first line of your program) and then add a 1 second sleep on the last line of the `while` loop.

Save the program and test it. As you walk around the game world, the program should create one block of water every second for 30 seconds. If you get stuck, go back to the steps in Mission #33 (page 125) for help.

Figure 7-2 shows the curse in action.

Figure 7-2: Oh no! I'm being followed by a small flood.

BONUS OBJECTIVE: A FASTER FLOOD

How would you make the loop repeat twice as fast (every half a second) while still lasting for 30 seconds?

INFINITE WHILE LOOPS

In most cases, it is very important that the Boolean condition in your `while` loop eventually become `False`; otherwise, the loop will iterate forever, and your computer might crash.

But there are times when you may want to program an infinite loop. For example, video games often use an infinite loop to check for user input and manage player movement. Of course, these video games include a Quit button so you can pause or stop the infinite loops when you need to take a break!

A simple way to create an infinite loop is to use a `True` condition when you define a `while` loop, as shown here:

```
while True:
    print("Hello")
```

This code will repeat forever, printing the string `"Hello"` over and over again. Whether or not you meant to create an infinite loop, pressing CTRL-C in the Python shell is a common way to stop it. In IDLE you can select **Shell ▸ Restart Shell** to stop the loop as well.

Note that any code that is placed *after* an infinite `while` loop will never run. In the following example, the last line of code is unreachable due to the infinite `while` loop that comes before it:

```
while True:
    print("Hello")
print("This line is never reached")
```

Although infinite loops can sometimes be tricky, you can also create them to do lots of cool things. Let's try this next!

MISSION #35: FLOWER TRAIL

The program you'll write in this mission is like the one in Mission #34, but instead of placing water blocks, you'll create a trail of flowers behind the player. Flowers are much nicer than floods!

Open the file *waterCurse.py* in the *whileLoops* folder and then save it as *flowerTrail.py*.

To make an infinite trail of flowers appear as the player walks around the game, make the following changes to the program:

1. Change the condition of the `while` loop to `True`.
2. Delete the `count` variable and the increment.
3. Change the block type argument in the `setBlock()` function from `8` to `38`.
4. Reduce the value of the argument in the `sleep()` function to `0.2` to make five flowers appear every second.
5. Save the program and run it. Figure 7-3 shows what you should see.

Figure 7-3: Look at all the beautiful flowers!

BONUS OBJECTIVE: A TRAIL OF DESTRUCTION

The *flowerTrail.py* program is very flexible. Try changing the block type that is placed by the program. A fun block type to try is explosive TNT (`setBlock(x, y, z, 46, 1)`). Notice the extra argument `1` after `46`, which is the TNT block type. The `1` sets the state of the TNT to make it detonate just by hitting it, without needing flint and steel. Just click the left mouse button a few times when pointing at the TNT to make it explode!

FANCY CONDITIONS

Because `while` loops expect a Boolean value for their condition, you can use any of the comparators and Boolean operators that you've learned about so far. For instance, you've already seen that the greater than and less than operators work just like they did in earlier chapters.

But you can control `while` loops with comparators and Boolean operators in other ways as well. Let's take a look!

We'll start by writing a more interactive condition. The following code creates the `continueAnswer` variable before the loop starts and checks that the value is equal to `"Y"`. Note that we can't use the word `continue` as a variable name because it is a reserved word in Python.

```
continueAnswer = "Y"
coins = 0
while continueAnswer == "Y":
    coins = coins + 1
    continueAnswer = input("Continue? Y/N")
print("You have " + str(coins) + " coins")
```

In the last line of the `while` loop, the program asks for input from the user. If the user presses anything besides `"Y"` in response, the loop will exit. The user can repeatedly press Y and Y and Y, and each time the value of the `coins` variable will increase by 1.

Notice that the variable being checked, `continueAnswer`, is created before the loop starts. If it wasn't, the program would display an error. That's why the variable we use to test the condition must exist before we try to use it, and it must be `True` when the program reaches the `while` loop the first time; otherwise, the condition won't be met, and the `while` loop's body statement will never execute.

MISSION #36: DIVING CONTEST

Let's have some fun with `while` loops and the equal to (`==`) comparator. In this mission, you'll create a mini-game in which the player dives underwater for as long as they can. The program will record how many seconds they stay underwater and display their score at the end of the program. To congratulate the player, the program will shower them with flowers if they stay underwater longer than 6 seconds.

Here is some code to get you started:

divingContest.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import time

score = 0
pos = mc.player.getPos()
❶ blockAbove = mc.getBlock(pos.x, pos.y + 2, pos.z)

❷ # Add a while loop here
time.sleep(1)
pos = mc.player.getPos()
❸ blockAbove = mc.getBlock(pos.x, pos.y + 2, pos.z)
❹ score = score + 1
mc.postToChat("Current score: " + str(score))

mc.postToChat("Final score: " + str(score))

❺ if score > 6:
    finalPos = mc.player.getTilePos()
    mc.setBlocks(finalPos.x - 5, finalPos.y + 10, finalPos.z - 5,
                 finalPos.x + 5, finalPos.y + 10, finalPos.z + 5, 38)
```

Save the program as *divingContest.py* in your *whileLoops* folder. The `score` variable keeps track of how many seconds the player is underwater.

Run the code to see what happens. At the moment, the program isn't complete: it only checks whether the player is underwater once and then finishes.

Before you fix this, let's look at what the rest of the code does. The `blockAbove` variable stores the type of the block located at the player's head ❶. For example, if the player's head is underwater, this variable will store a value of 8 (which means the block is water). Later in the code, you'll set `blockAbove` to store the value of the block above the player's head again ❸ so when you create your `while` loop, it will update `blockAbove` to the current block above the player's head. At ❹, the program adds 1 point to the total for every second the player is underwater, and at ❺, it uses an `if` statement to create a shower of flowers above the player if the score is greater than 6.

It's up to you to add a loop to the program that uses the `blockAbove` variable as a condition at ❷. Make the `while` loop check whether `blockAbove` is equal to water (block type 8) or equal to flowing water (block type 9). You can use the following condition in the while loop to check this: `while blockAbove == 8 or blockAbove == 9`. This checks whether the player is currently underwater and will continue to check whether the player is underwater every time the loop repeats.

To test your program, find some water that's at least three blocks deep and dive into it. The program will run only if you're already underwater. When you run the program, it should start displaying how many seconds you've been underwater. After a while, swim to the surface. The program should display your score and shower you with flowers if you were underwater for 6 seconds or more. Figure 7-4 shows the player underwater and the score being displayed. Figure 7-5 shows the flowers that appear when you win.

Figure 7-4: I'm holding my breath underwater, and the number of seconds I've been underwater is displayed.

Figure 7-5: I won my very own flowery celebration!

BONUS OBJECTIVE: A WINNER IS YOU

Try adding extra prizes by writing more code in the `if` statement at the end of the program. If the player gets a high score, you could give them a gold block. Try adding several levels of difficulty with different prizes for each one.

BOOLEAN OPERATORS AND WHILE LOOPS

You can use Boolean operators like *and*, *or*, and *not* with a `while` loop when you want the loop to use more than one condition. For example, the following loop will iterate while the user has not input the correct password and has made three attempts or fewer:

```
password = "cats"
passwordInput = input("Please enter the password: ")
attempts = 0

❶ while password != passwordInput and attempts < 3:
❷     attempts += 1
❸     passwordInput = input("Incorrect. Please enter the password: ")

❹ if password == passwordInput:
      print("Password accepted.")
```

The `while` loop condition ❶ does two tasks: it checks whether the password is different from the user's input (`password != passwordInput`) and checks whether the user has tried to enter the password three times or less (`attempts < 3`). The `and` operator allows the `while` loop to check both conditions at the same time. If the condition is `False`, the loop increments the

attempts variable ❷ and asks the user to reenter the password ❸. The loop will finish if the user enters the correct password or the attempts variable is greater than 3. After the loop finishes, the program will output Password accepted only if the user entered the correct password ❹.

CHECKING A RANGE OF VALUES IN WHILE LOOPS

You can also check for values in a certain range using a while loop. For example, the following code checks whether the value the user has entered is between 0 and 10. If it is not, the loop will exit.

```
position = 0
❶ while 0 <= position <= 10:
    position = int(input("Enter your position 0-10: "))
    print(position)
```

If the position variable is greater than 10, the loop won't repeat ❶. The same will happen if the value is less than 0. This is useful in Minecraft when you're checking whether the player's position is in a certain area in the game, as you'll see in the next mission.

MISSION #37: MAKE A DANCE FLOOR

It's time to dance! But before you can bust out some sweet moves, you'll need a dance floor. The program in this mission will generate a dance floor that flashes different colors every half second as long as the player stays on the floor.

The following is the start of the code. It creates a dance floor at the player's current position and uses an if statement to change colors. But the code is not complete.

danceFloor.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import time

pos = mc.player.getTilePos()
floorX = pos.x - 2
floorY = pos.y - 1
floorZ = pos.z - 2
width = 5
length = 5
block = 41
❶ mc.setBlocks(floorX, floorY, floorZ,
               floorX + width, floorY, floorZ + length, block)

❷ while floorX <= pos.x <= floorX + width and  # Check z is within the floor
❸     if block == 41:
          block = 57
      else:
          block = 41
```

```
    mc.setBlocks(floorX, floorY, floorZ,
                 floorX + width, floorY, floorZ + length, block)
    pos = mc.player.getTilePos()
    time.sleep(0.5)
```

Open IDLE, create a new file, and save the program as *danceFloor.py* in the *whileLoops* folder. The code builds the dance floor based on the player's current position ❶ and stores the dance floor's location and size in the `floorX`, `floorY`, `floorZ`, `width`, and `length` variables. Inside the `while` loop, the code uses an `if` statement to alternate the blocks that the dance floor is made of ❸, making the dance floor look like it's flashing.

To get the program to work properly, you need to change the `while` loop's condition to check whether the player's z-coordinate is on the dance floor ❷. In other words, check whether `pos.z` is greater than or equal to `floorZ` and less than or equal to `floorZ` plus `length`. For guidance, look at how I checked whether `pos.x` is on the dance floor by using (`floorX <= pos.x <= floorX + width`). Figure 7-6 shows the dance floor in action!

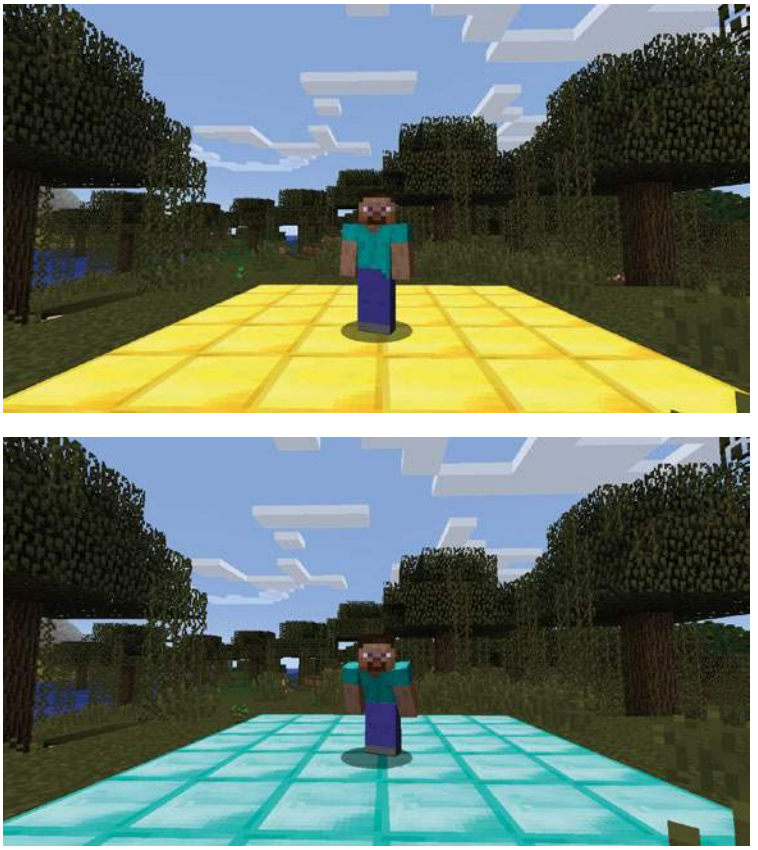

Figure 7-6: I'm showing off my moves on the dance floor.

When you've completed the program, save it and run it. A dance floor should appear below the player and change every half second. Dance

around a bit—have some fun! When you're done, leave the dance floor and make sure it stops flashing. It won't switch on again unless you run the program again to create a new dance floor.

BONUS OBJECTIVE: PARTY'S OVER

When the player is finished dancing on the dance floor, make the floor disappear. To do this, change the dance floor to air when the loop finishes.

NESTED IF STATEMENTS AND WHILE LOOPS

You can write more powerful programs by using `if` statements and nested `if` statements inside `while` loops. You may have noticed a nested `if` statement in the code in Mission #37 (page 135).

In the following example, the nested `if` statement checks the last word that was printed and decides whether to print the words `"mine"` and `"craft"`. The loop repeats 50 times.

```
word = "mine"
count = 0
while count < 50:
    print(word)
    if word == "mine":
        word = "craft"
    else:
        word = "mine"
```

The `word` variable stores the first word that will be printed. The `if` statement in the loop checks whether the current word is `"mine"` and, if it is, changes the word to `"craft"` and prints it on the next iteration of the loop. If the word isn't `"mine"`, it will be changed to `"mine"`. This is an infinite loop, so be sure to use CTRL-C to escape!

You can also nest `elif` statements and other `while` loops inside `while` loops.

The following program asks the user if they want to print all the numbers between one and a million:

```
userAnswer = input("Print the numbers between 1 and 1000000? (yes/no): ")

❶ if userAnswer = "yes":
      count = 1
❷     while count <= 1000000:
          print(count)
          count += 1
```

The `if` statement checks whether the user's input is `yes` ❶. If it is, the program runs the loop that is nested in the `if` statement ❷. If the input is anything else, the program won't run the loop and will finish.

MISSION #38: THE MIDAS TOUCH

Midas is a king of legend. Everything he touched turned to gold. Your mission is to write a program that changes every block below the player to gold—except for air and water, of course, or you'd be in real trouble! Recall that the gold block has a value of 41, still water is 9, and air is 0.

midas.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

air = 0
water = 9

❶ # Add an infinite while loop here
    pos = mc.player.getTilePos()
    blockBelow = mc.getBlock(pos.x, pos.y - 1, pos.z)

❷     # Add if statement here
        mc.setBlock(pos.x, pos.y - 1, pos.z, 41)
```

Open IDLE and create a new file. Save the file as *midas.py* in the *whileLoops* folder. You need to add a bit more to the program so it can do what you need it to do. First, you'll add an infinite `while` loop ❶. Remember that an infinite `while` loop has a condition that is always `True`. You also need to add an `if` statement that checks whether the block below the player is not equal to air and not equal to still water ❷. The value of the block below the player is stored in the `blockBelow` variable, and the values for air and water are stored in the `air` and `water` variables.

When you've completed the program, save it and run it. The player should leave a trail of gold behind them. When you jump in water or fly in the air, the blocks below you should not change. Figure 7-7 shows the program in action.

Figure 7-7: Every block I walk on turns to gold.

To exit the infinite loop, go to **Shell ▸ Restart Shell** in your IDLE shell or click in the shell and press CTRL-C.

BONUS OBJECTIVE: I'M A PLOWMAN

You can change *midas.py* to serve a variety of purposes. How would you change it so it automatically changes dirt blocks to hoed farmland? How about changing dirt blocks to grass blocks?

ENDING A WHILE LOOP WITH BREAK

With `while` loops, you have complete control over how and when the loop ends. So far you've only used conditions to end loops, but you can also use a `break` statement. The `break` statement lets your code immediately exit a `while` loop. Let's look at this concept!

One way to use `break` statements is to put them in an `if` statement nested in the loop. Doing so immediately stops the loop when the `if` statement's condition is `True`. The following code continually asks for user input until they type `"exit"`:

```
❶ while True:
❷     userInput = input("Enter a command: ")
❸     if userInput == "exit":
❹         break
      print(userInput)
❺ print("Loop exited")
```

This is an infinite loop because it uses `while True` ❶. Each time the loop repeats, it asks for the user to enter a command ❷. The program checks whether the input is `"exit"` ❸ using an `if` statement. If the input meets the condition, the `break` statement stops the loop from repeating ❹, and the program continues on the line immediately after the body of the loop, printing `"Loop exited"` to the Python shell ❺.

MISSION #39: CREATE A PERSISTENT CHAT WITH A LOOP

In Mission #13 (page 72), you created a program that posts the user's message to chat using strings, input, and output. Although this program was useful, it was quite limited because you had to rerun the program every time you wanted to post a new message.

In this mission, you'll improve your chat program using a `while` loop so users can post as many messages as they want without restarting the program.

Open the *userChat.py* file in the *strings* folder and then save it as *chatLoop.py* in the *whileLoops* folder.

To post a new message every time you want to without rerunning the program, add the following to your code:

1. Add an infinite `while` loop to the program.
2. Add an `if` statement to the loop to check whether the user's input is `"exit"`. If the input is `"exit"`, the loop should break.
3. Make sure the `userName` variable is defined before the start of the loop.

When you've added the changes, save your program and run it. A prompt in the Python shell will ask you to type in a username. Do this and press ENTER. The program will then ask you to enter a message. Type a message and then press ENTER. The program will keep asking you to enter a message until you type `exit`. Figure 7-8 shows my chat program running.

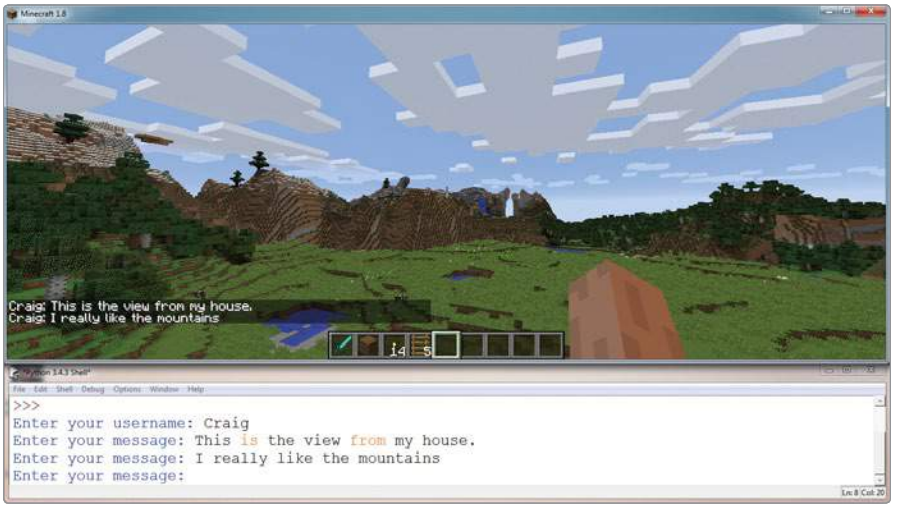

Figure 7-8: I'm chatting with myself.

BONUS OBJECTIVE: BLOCK CHAT

Expand the chat feature so users can create blocks. For example, if the user enters `"wool"`, the program creates a wool block. You can do this by adding `elif` statements to your `if` statement to check user input.

WHILE-ELSE STATEMENTS

Like an `if` statement, `while` loops can have secondary conditions triggered by `else` statements.

The `else` statement executes when the condition of a `while` statement is `False`. Unlike the body of a `while` statement, the `else` statement will execute only once, as shown here:

```
message = input("Please enter a message.")

while message != "exit":
    print(message)
    message = input("Please enter a message.")
else:
    print("User has left the chat.")
```

This loop repeats as long as the `message` entered is not equal to `"exit"`. If the `message` is `"exit"`, the loop will stop repeating, and the body of the `else` statement will print `"User has left the chat."`

If you use a `break` statement in the `while` statement, the `else` isn't executed. The following code is similar to the preceding example but includes a nested `if` statement and a `break` statement. When the user types `abort` instead of `exit`, the chat loop will exit without printing the `"User has left the chat."` message to the chat.

```
message = input("Please enter a message.")

while message != "exit":
    print(message)
    message = input("Please enter a message.")
    if message == "abort":
        break
else:
    print("User has left the chat.")
```

The `if` statement checks whether the message entered is `"abort"`. If this is `True`, the `break` statement runs and the loop will exit. Because the `break` statement was used, the body of the `else` statement will not run, and `"User has left the chat."` will not be printed.

MISSION #40: HOT AND COLD

In this mission, we'll create a Hot and Cold game in Minecraft. If you've never played, the idea is that your friend hides an object and you have to find it. Your friend gives you hints based on how far away from the object you are. If you're close, your friend says "Hot," and if you're far away, they'll say "Cold." When you're right next to the object, they'll say "You're on fire!" and if you're very far away, they'll say "Freezing!"

The object of the game is to find and stand on the diamond block that has been placed randomly in the game world. In this version of the game, you'll play by yourself, and the Python program will tell you how far away from the hidden block you are. The game ends when you stand on the diamond block.

Listing 7-2 places a block in a random location.

blockHunter.py

```
from mcpi.minecraft import Minecraft
import math
import time
import random
mc = Minecraft.create()

destX = random.randint(-127, 127)
destZ = random.randint(-127, 127)
❶ destY = mc.getHeight(destX, destZ)

block = 57
❷ mc.setBlock(destX, destY, destZ, block)
mc.postToChat("Block set")

while True:
    pos = mc.player.getPos()
❸   distance = math.sqrt((pos.x - destX) ** 2 + (pos.z - destZ) ** 2)

❹   if distance > 100:
        mc.postToChat("Freezing")
    elif distance > 50:
        mc.postToChat("Cold")
    elif distance > 25:
        mc.postToChat("Warm")
    elif distance > 12:
        mc.postToChat("Boiling")
    elif distance > 6:
        mc.postToChat("On fire!")
    elif distance == 0:
❺       mc.postToChat("Found it")
```

Listing 7-2: The start of the Hot and Cold program

Before randomly placing a block, the program makes sure that the block won't be placed underground. To do so, it uses the `getHeight()` function ❶, which finds the block that is the highest y-coordinate (that is, on the surface) for any position in the game. Then it places a diamond block at a random position ❷.

The code at ❸ calculates the distance to the diamond block. It uses the `sqrt()` function, which is in the `math` module—this is why `import math` is needed at the beginning of the program. The `sqrt()` function calculates the square root of a number.

NOTE *Listing 7-2 uses a formula called the* Pythagorean theorem. *The formula uses two sides of a triangle to calculate the length of the third. In this case, I use the distance from the player to the hidden block on the x-axis and the z-axis to calculate the distance to the hidden block in a straight line.*

The message that the program displays depends on how far away you are from the block, which you can find out using an `if` statement and the `distance` variable ❹. The program displays `"Freezing"` if you're very far away and `"On fire!"` if you're very close.

Copy Listing 7-2 into a new file in IDLE and save the program as *blockHunter.py* in the *whileLoops* folder.

At the moment the program works, but it doesn't end when you find the block. To finish the code, you need to add a `break` statement when the player's distance from the block is 0 ❺.

When you've completed the program, save it and run it. A random block will be generated, and you'll need to find it. The program should stop when you find the block and stand on it. Figure 7-9 shows that I've just found the block.

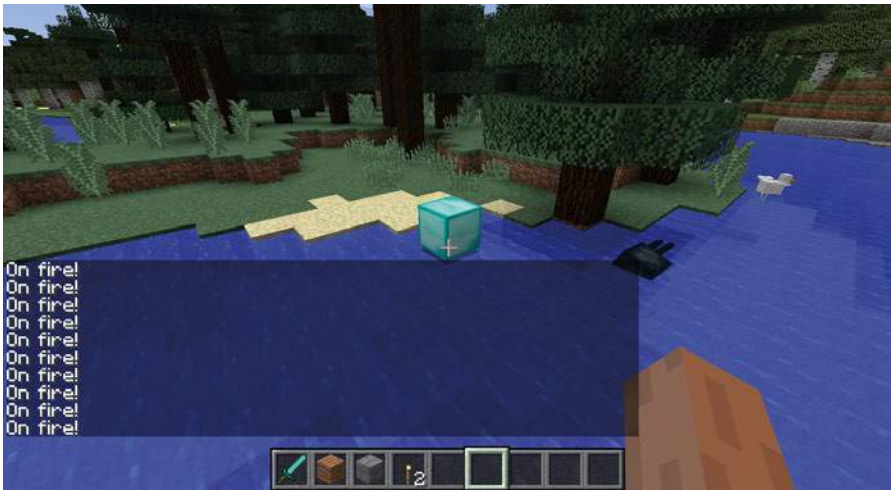

Figure 7-9: I've found the block, and now I just need to stand on it.

BONUS OBJECTIVE: TIME FOR TIME

The *blockHunter.py* program gives you as long as you need to find the block. Can you think of a way to display how long it takes the player to find the block or even limit the amount of time they have to play the game?

WHAT YOU LEARNED

Well done! You've learned a lot about `while` loops. You can create `while` loops and infinite `while` loops, and you can use loops with conditions and Boolean operators. Using loops, you can now write programs that repeat code, which will save you lots of time so you can focus on mastering Minecraft. In Chapter 8, you'll learn another way to make reusable code using functions.