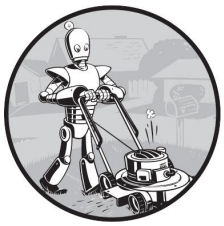
3

LOOPS



In the previous chapter, you learned how to make programs run certain blocks of code while skipping others. But there's more to flow control than this. In this chapter, you'll learn how to repeatedly execute blocks of code using loops. Python's two kinds of loops, `while` and `for`, open up the full power of automation, because they can run lines of code millions of times per second. You'll also learn how to import code libraries, called *modules*, to make even more functions available to your programs.

while Loop Statements

You can make a block of code execute over and over again using a `while` statement. The code in a `while` clause will be executed as long as the statement's condition is `True`. In code, a `while` statement always consists of the following:

- The `while` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `while` clause or `while` block)

You can see that a `while` statement looks similar to an `if` statement. The difference is in how they behave. At the end of an `if` clause, the program execution continues after the `if` statement. But at the end of a `while` clause, the program execution jumps back to the

start of the `while` statement. The `while` clause is often called the *while loop* or just the *loop*.

Let's look at an `if` statement and a `while` loop that use the same condition and take the same actions based on that condition. Here is the code with an `if` statement:

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Here is the code with a `while` statement:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

These statements are similar; both `if` and `while` check the value of `spam`, and if it's less than `5`, they print a message. But when you run these two code snippets, something very different happens for each one. For the `if` statement, the output is simply `"Hello, world."` But for the `while` statement, it's `"Hello, world."` repeated five times! Take a look at the flowcharts for these two pieces of code, Figures 3-1 and 3-2, to see why this happens.

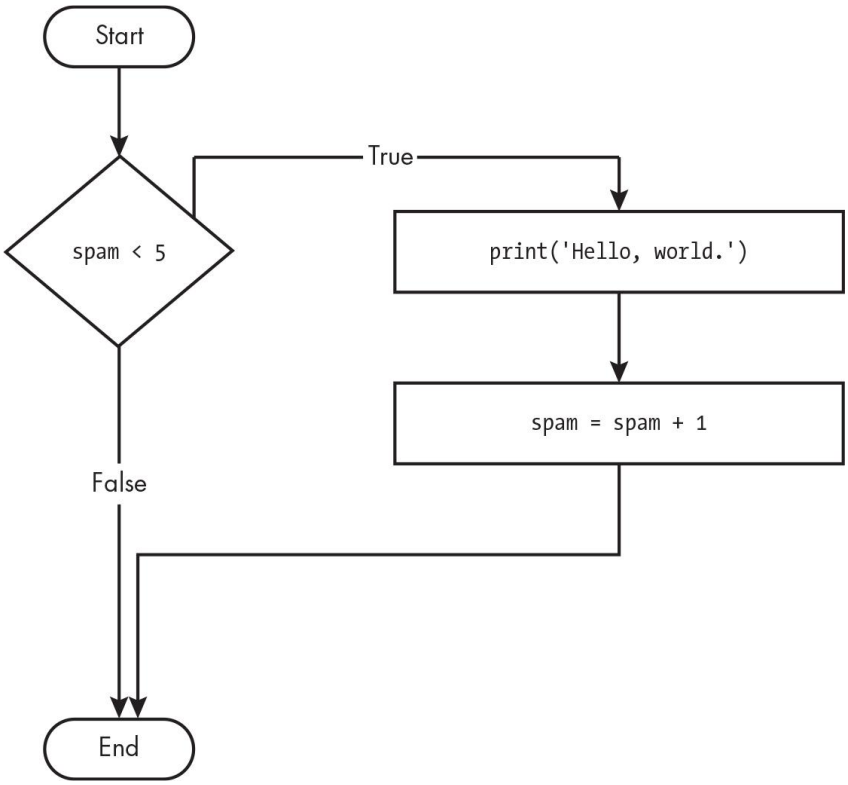


Figure 3-1: The flowchart for the if statement code Description

The code with the `if` statement checks the condition, and it prints `"Hello, world."` only once if that condition is true. The code with the `while` loop, on the other hand, will print it five times. The loop stops after five prints because the integer in `spam` increases by one at the end of each loop iteration, which means that the loop will execute five times before `spam < 5` is `False`.

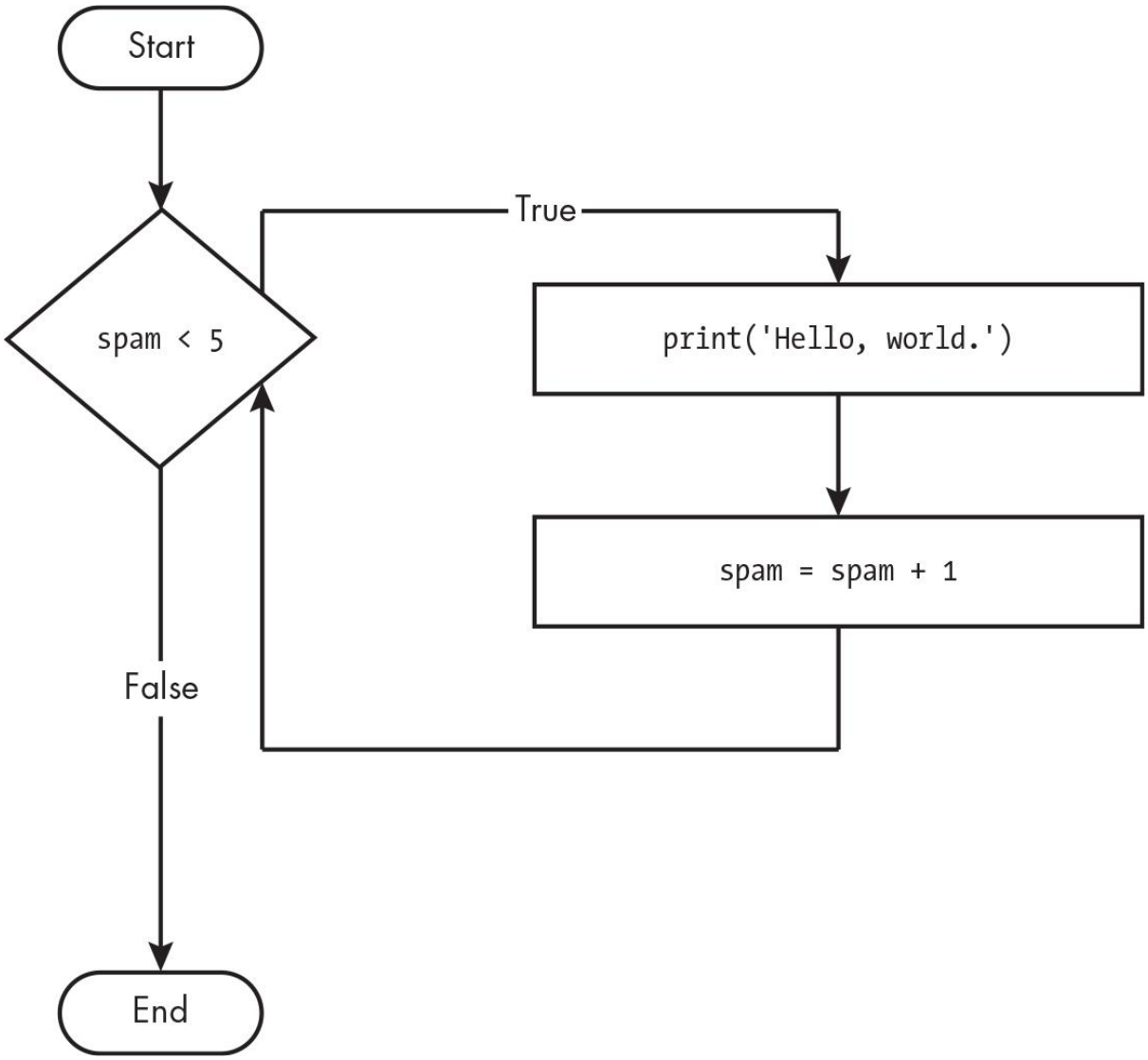


Figure 3-2: The flowchart for the while statement code Description

In the `while` loop, the condition is always checked at the start of each *iteration* (that is, each time the loop is executed). If the condition is `True`, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be `False`, the `while` clause is skipped.

An Annoying while Loop

Here's a small example program that will keep asking you to type, literally, **`your name`**. Select **File ▸ New** to open a new file editor window, enter the following code, and save the file as *yourName.py*:

```
name = ''
while name != 'your name':
    print('Please type your name.')
    name = input('>')
print('Thank you!')
```

First, the program sets the `name` variable to an empty string. This is so that the `name != 'your name'` condition will evaluate to `True` and the program execution will enter the `while` loop's clause.

The code inside this clause asks the user to type their name, which is assigned to the `name` variable. Since this is the last line of the block, the execution moves back to the start of the `while` loop and reevaluates the condition. If the value in `name` is *not* equal to the string `'your name'`, the condition is `True`, and the execution enters the `while` clause again.

But once the user literally enters `your name`, the condition of the `while` loop will be `'your name' != 'your name'`, which evaluates to `False`. Now, instead of the program execution reentering the `while` loop's clause, Python skips past it and continues running the rest of the program. Figure 3-3 shows the flowchart for the *yourName.py* program.

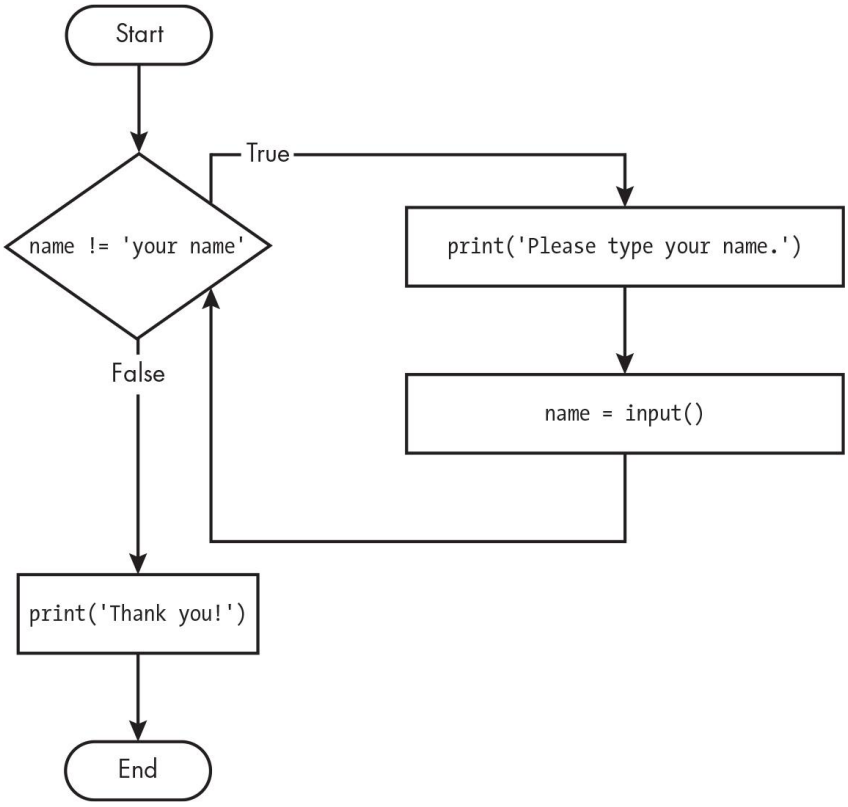


Figure 3-3: The flowchart for the yourName.py *program* Description

Now let's see *yourName.py* in action. Press F5 to run it, and enter something other than `your name` a few times before you give the program what it wants:

```
Please type your name.
>Al
Please type your name.
```

```
>Albert
Please type your name.
>%#@#%*(^&!!!
Please type your name.
>your name
Thank you!
```

If you never enter `your name`, then the `while` loop's condition will never be `False`, and the program will just keep asking you the question forever. Here, the `input()` call lets the user enter the right string to make the program move on. In other programs, the condition might never actually change, and that can be a problem. Let's look at how you can break out of a `while` loop.

break Statements

There is a shortcut to getting the program execution to break out of a `while` loop's clause early. If the execution reaches a `break` statement, it immediately exits the `while` loop's clause. In code, a `break` statement simply contains the `break` keyword.

Here's a program that does the same thing as the previous *yourName.py* program but uses a `break` statement to escape the loop. Enter the following code, and save the file as *yourName2.py*:

```
while True:
    print('Please type your name.')
    name = input('>')
    if name == 'your name':
        break
print('Thank you!')
```

The first line creates an *infinite loop*; it is a `while` loop whose condition is always `True`. (The expression `True`, after all, always evaluates to the value `True`.) After the program execution enters this loop, it will exit the loop only when a `break` statement is executed. (An infinite loop that *never* exits is a common programming bug.)

Just like before, this program asks the user to enter `your name`. Now, however, while the execution is still inside the `while` loop, an `if` statement checks whether `name` is equal to `'your name'`. If this

condition is `True`, the `break` statement is run, and the execution moves out of the loop to `print('Thank you!')`. Otherwise, the `if` statement clause that contains the `break` statement is skipped, which puts the execution at the end of the `while` loop. At this point, the program execution jumps back to the start of the `while` statement to recheck the condition. Since this condition is merely the `True` Boolean value, the execution enters the loop to ask the user to enter `your name` again. See Figure 3-4 for this program's flowchart.

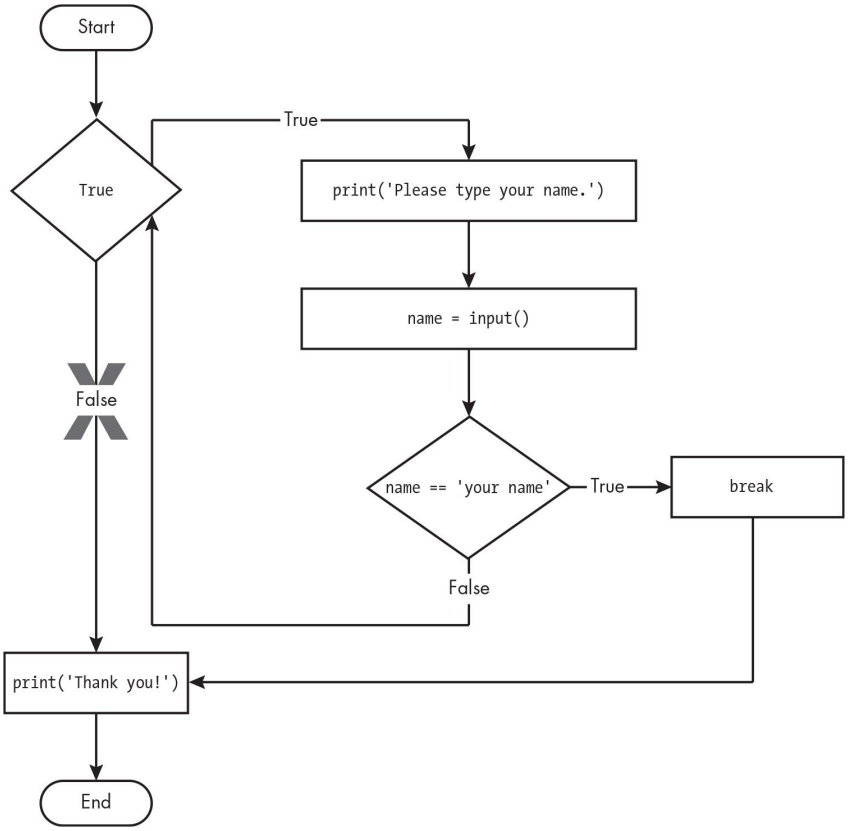


Figure 3-4: The flowchart for the yourName2.py *program with an infinite loop. Note that the X path will logically never happen, because the loop condition is always* `True`. Description

Run *yourName2.py*, and enter the same text you entered for *yourName.py*. The rewritten program should respond in the same way as the original.

continue Statements

Like `break` statements, we use `continue` statements inside loops. When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition. (This is also what happens when the execution reaches the end of the loop.)

TRAPPED IN AN INFINITE LOOP?

If you ever run a program that has a bug causing it to get stuck in an infinite loop, press CTRL-C. This will send a `KeyboardInterrupt` error to your program and cause it to stop immediately. You can also click the Stop button at the top of the Mu window. Try stopping a program by creating a simple infinite loop in the file editor, and save the program as *infiniteLoop.py*. If you are running the program from Mu, you can also click the Stop button:

```
while True:
    print('Hello, world!')
```

When you run this program, it will print `Hello, world!` to the screen forever because the `while` statement's condition is always `True`. CTRL-C is also handy if you want to simply terminate your program immediately, even if it's not stuck in an infinite loop.

Let's use `continue` to write a program that asks for a name and password. Enter the following code into a new file editor window and save the program as *swordfish.py*:

```
while True:
    print('Who are you?')
    name = input('>')
 ❶ if name != 'Joe':
     ❷ continue
    print('Hello, Joe. What is the password? (It is a fish.)')
 ❸ password = input('>')
    if password == 'swordfish':
        ❹ break
❺ print('Access granted.')
```

If the user enters any name besides `Joe` ❶, the `continue` statement ❷ causes the program execution to jump back to the start of the loop. When the program reevaluates the condition, the execution will always enter the loop, because the condition is simply the value `True`. Once the user makes it past that `if` statement, they are asked for

a password ❸. If the password entered is `swordfish`, the `break` statement ❹ is run, and the execution jumps out of the `while` loop to print `Access granted.` ❺ Otherwise, the execution continues to the end of the `while` loop, where it then jumps back to the start of the loop. See Figure 3-5 for this program's flowchart.

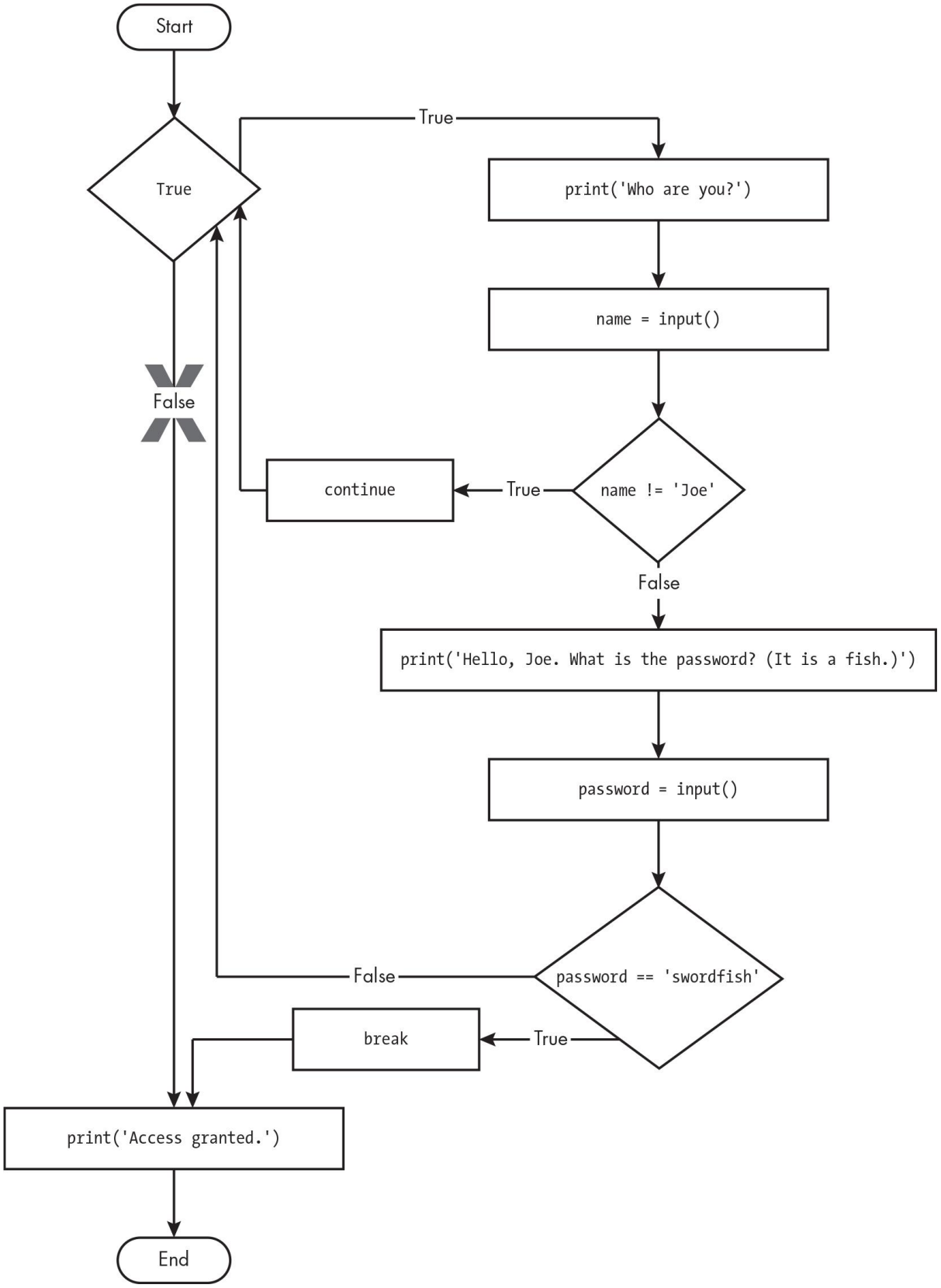


Figure 3-5: The flowchart for the swordfish.py *program. The X path will logically never happen, because the loop condition is always* `True`. Description

Run this program and give it some input. Until you claim to be Joe, the program shouldn't ask for a password, and once you enter the correct password, it should exit:

```
Who are you?
>I'm fine, thanks. Who are you?
Who are you?
>Joe
Hello, Joe. What is the password? (It is a fish.)
>Mary
Who are you?
>Joe
Hello, Joe. What is the password? (It is a fish.)
>swordfish
Access granted.
```

"TRUTHY" AND "FALSEY" VALUES AND THE BOOL() FUNCTION

Conditions will consider some values in other data types equivalent to `True` or `False`. When used in conditions, `0`, `0.0`, and `''` (the empty string) are considered `False`, while all other values are considered `True`. For example, look at the following program:

```
name = ''
❶ while not name:
    print('Enter your name:')
    name = input('>')
print('How many guests will you have?')
num_of_guests = int(input('>'))
❷ if num_of_guests:

❸ print('Be sure to have enough room for all
```

```
your guests.')
print('Done')
```

If the user enters a blank string for `name`, the `while` statement's condition will be `True` ❶, and the program will continue to ask for a name. If the value for `num_of_guests` is not 0 ❷, the condition is considered to be `True`, and the program will print a reminder for the user ❸.

You could have entered `not name != ''` instead of `not name`, and `num_of_guests != 0` instead of `num_of_guests`, but using the truthy and falsey values can make your code easier to read.

If you want to know if a value is truthy or falsey, pass it to the `bool()` function as in this interactive shell example:

```
>>> bool(0)
False
>>> bool(42)
True
>>> bool('Hello')
True
>>> bool('')
False
```

for Loops and the range() Function

The `while` loop keeps looping while its condition is `True` (which is the reason for its name), but what if you want to execute a block of code only a certain number of times? You can do this with a `for` loop statement and the `range()` function.

In code, a `for` statement looks something like `for i in range(5):` and includes the following:

- The `for` keyword
- A variable name
- The `in` keyword

- A call to the `range()` function with up to three integers passed to it
- A colon
- Starting on the next line, an indented block of code (called the `for` clause or `for` block)

Let's create a new program called *fiveTimes.py* to help you see a `for` loop in action:

```
print('Hello!')
for i in range(5):
    print('On this iteration, i is set to ' +
str(i))
print('Goodbye!')
```

The code in the `for` loop's clause is run five times. The first time it is run, the variable `i` is set to `0`. The `print()` call in the clause will print `On this iteration, i is set to 0`. After Python finishes an iteration through all the code inside the `for` loop's clause, the execution goes back to the top of the loop, and the `for` statement increments `i` by one. This is why `range(5)` results in five iterations through the clause, with `i` being set to `0`, then `1`, then `2`, then `3`, and then `4`. The variable `i` will go up to, but will not include, the integer passed to `range()`. Figure 3-6 shows the flowchart for the *fiveTimes.py* program.

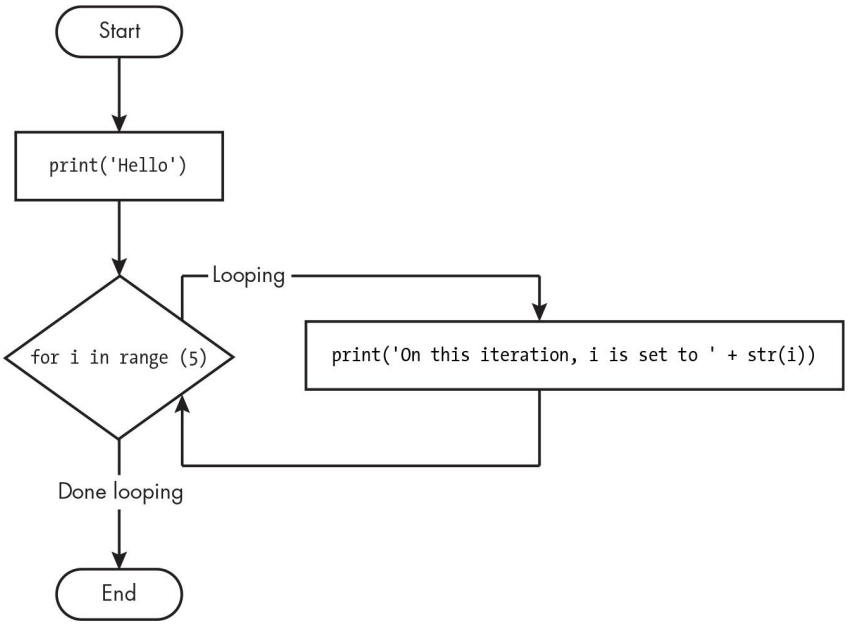


Figure 3-6: The flowchart for the fiveTimes.py *program* Description

Here is the complete output of the program:

```
Hello!
On this iteration, i is set to 0
On this iteration, i is set to 1
On this iteration, i is set to 2
On this iteration, i is set to 3
On this iteration, i is set to 4

Goodbye!
```

You can use `break` and `continue` statements inside `for` loops as well. The `continue` statement will continue to the next value of the `for` loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use `continue` and `break` statements only inside `while` and `for` loops. If you try to use these statements elsewhere, Python will give you an error.

As another `for` loop example, consider this story about the mathematician Carl Friedrich Gauss. When Gauss was a boy, a teacher wanted to give the class some busywork. The teacher told them to add up all the numbers from 0 to 100. Young Gauss came up with a clever trick to figure out the answer in a few seconds, but you can write a Python program with a `for` loop to do this calculation for you:

```
total = 0
for num in range(101):
    total = total + num
print(total)
```

The result should be 5,050. When the program first starts, the `total` variable is set to `0`. The `for` loop then executes `total = total + num` 101 times, each time with an incremented `num` variable. By the time the loop has finished all of its 101 iterations, every integer from `0` to `100` will have been added to `total`. At this point, `total` is printed to the screen. Even on the slowest computers, this program takes less than a second to complete.

(Young Gauss figured out a way to solve the problem in seconds. There are 50 pairs of numbers that add up to 101: 1 + 100, 2 + 99, 3 +

98, and so on, until 50 + 51. Since 50 × 101 is 5,050, the sum of all the numbers from 0 to 100 is 5,050. Clever kid!)

WHY "UP TO BUT NOT INCLUDING"?

It may seem strange that the `range()` function in `for` loops has you specify the number that the loop goes up to, minus one. In programming, ranges are often specified in a "closed, open" format that includes the starting number but excludes the ending number. For example, the range 0, 10 would include the numbers 0 to 9 instead of 0 to 10. There are many advantages of using "closed, open" instead of "closed, closed" that lead to fewer bugs. Calculating the size of the range is just a matter of subtracting the starting number from the ending number. The 0, 10 range (which has the numbers 0 to 9 instead of 0 to 10) has 10 – 0 or 10 numbers. With a "closed, closed" range of 0, 9, you have to calculate the length as 9 – 0 + 1. (And it's easy to mistakenly do *off-by-one* errors like 9 – 0 – 1.) The start of the next range 10, 20 is just a matter of using the previous ending number as the new starting number.

For programs that deal with timestamps, the "closed, open" range of 00:00:00, 24:00:00.0 is much easier to work with than the "closed, closed" range of 00:00:00, 23:59:59.999. "Closed, open" ranges may seem odd at first, but they'll become second nature as you get more experience writing code.

An Equivalent while Loop

You can actually use a `while` loop to do the same thing as a `for` loop; `for` loops are just more concise. Let's rewrite *fiveTimes.py* to use a `while` loop equivalent of a `for` loop:

```
print('Hello!')
i = 0
while i < 5:
    print('On this iteration, i is set to ' +
str(i))
    i = i + 1
print('Goodbye!')
```

If you run this program, the output should look the same as the *fiveTimes.py* program, which uses a `for` loop. Remember that `for` loops are useful for looping a specific number of times, and `while` loops are useful for looping as long as a particular condition is true.

Arguments to range()

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence of integers, including starting at a number other than zero:

```
for i in range(12, 16):
    print(i)
```

The first argument will be where the `for` loop's variable starts, and the second argument will be up to, but not including, the number to stop at:

```
12
13
14
15
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the *step argument*. The step is the amount by which the variable is increased after each iteration:

```
for i in range(0, 10, 2):
    print(i)
```

So, calling `range(0, 10, 2)` will count from zero to eight by intervals of two:

```
0
2
4
```

```
6
8
```

The `range()` function is flexible in the sequence of numbers it produces for `for` loops. *For* example (I never apologize for my puns), you can even use a negative number for the step argument to make the `for` loop count down instead of up:

```
for i in range(5, -1, -1):
    print(i)
```

This `for` loop would have the following output:

```
5
4
3
2
1
0
```

Running a `for` loop to print `i` with `range(5, -1, -1)` should print the numbers from five down to zero.

Importing Modules

All Python programs can call a basic set of functions called *built-in functions*, including the `print()`, `input()`, and `len()` functions you've seen before. Python also comes with a set of modules called the *standard library*. Each module is a Python program that contains a related group of functions that can be embedded in your programs. For example, the `math` module has mathematics-related functions, the `random` module has random number–related functions, and so on.

Before you can use the functions in a module, you must import the module with an `import` statement. In code, an `import` statement consists of the following:

- The `import` keyword
- The name of the module

- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

Enter this code into the file editor, and save it as *printRandom.py*:

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

When you run this program, the output will look something like this:

```
4
1
8
4
1
```

The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it. Because `randint()` is in the `random` module, you must first enter **`random.`** in front of the function name to tell Python to look for this function inside the `random` module.

DON'T OVERWRITE MODULE NAMES

When you save your Python scripts, take care not to give them a name that is used by one of Python's modules, such as *random.py*, *sys.py*, *os.py*, or *math.py*. If you accidentally name one of your programs, say, *random.py*, and use an `import random` statement in another program, your program will import your *random.py* file instead of Python's `random` module. This can lead to errors such as `AttributeError: module 'random' has no attribute 'randint'`, since your *random.py* file doesn't have the functions that the real `random` module has. Don't use the names of any built-in Python functions for your file or variable names, either. Some common Python names are `all`, `any`, `date`, `email`, `file`, `format`,

hash, id, input, list, min, max, object, open, random, set, str, sum, test, and type.

Here's an example of an `import` statement that imports four different modules:

```
import random, sys, os, math
```

Now we can use any of the functions in these four modules. You'll learn more about them later in the book.

An alternative form of the `import` statement is composed of the `from` keyword, followed by the module name, the `import` keyword, and a star (`*`); for example, `from random import *`. With this form of `import` statement, calls to functions in `random` won't need the `random.` prefix. However, using the full name makes for more readable code, so it is better to use the `import random` form of the statement.

Ending a Program Early with sys.exit()

The last flow control concept to cover is how to *terminate*, or exit, the program. Programs always terminate if the program execution reaches the bottom of the instructions. However, you can cause the program to terminate before the last instruction by calling the `sys.exit()` function. Since this function is in the `sys` module, you have to import `sys` before your program can use it.

Open a file editor window and enter the following code, saving it as *exitExample.py*:

```
import sys

while True:
    print('Type exit to exit.')
    response = input('>')
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

Run this program in your code editor. This program has an infinite loop with no `break` statement inside. The only way this program will end is if the execution reaches the `sys.exit()` call. When `response` is equal to `'exit'`, the line containing the `sys.exit()` call is executed. Since the `response` variable is set by the `input()` function, the user must enter `exit` in order to stop the program.

A Short Program: Guess the Number

The examples I've shown you so far are useful for introducing basic concepts, but now you'll see how everything you've learned comes together in a more complete program. In this section, I'll show you a simple guess the number game. When you run this program, the output will look something like this:

```
I am thinking of a number between 1 and 20.
Take a guess.
>10
Your guess is too low.
Take a guess.
>15
Your guess is too low.
Take a guess.
>17
Your guess is too high.
Take a guess.
>16
Good job! You got it in 4 guesses!
```

Enter the following source code into the file editor, and save the file as *guessTheNumber.py*:

```
# This is a guess the number game.
import random
secret_number = random.randint(1, 20)
print('I am thinking of a number between 1
and 20.')
```

```
# Ask the player to guess 6 times.
for guesses_taken in range(1, 7):
    print('Take a guess.')
    guess = int(input('>'))

    if guess < secret_number:
        print('Your guess is too low.')
    elif guess > secret_number:
        print('Your guess is too high.')
    else:
        break  # This condition is the
correct guess!

if guess == secret_number:
    print('Good job! You got it in ' +
str(guesses_taken) + ' guesses!')
else:
    print('Nope. The number was ' +
str(secret_number))
```

Let's look at this code line by line, starting at the top:

```
# This is a guess the number game.
import random
secret_number = random.randint(1, 20)
```

First, a comment at the top of the code explains what the program does. Then, the program imports the `random` module so that it can use the `random.randint()` function to generate a number for the user to guess. The return value, a random integer between 1 and 20, is stored in the variable `secret_number`:

```
print('I am thinking of a number between 1
and 20.')
```

```
# Ask the player to guess 6 times.
for guesses_taken in range(1, 7):
    print('Take a guess.')
    guess = int(input('>'))
```

The program tells the player that it has come up with a secret number and will give the player six chances to guess it. The code lets the player enter a guess and checks that guess in a `for` loop that will loop at most six times. The first thing that happens in the loop is that the player types in a guess. Because `input()` returns a string, its return value is passed straight into `int()`, which translates the string into an integer value. This gets stored in a variable named `guess`:

```
    if guess < secret_number:
        print('Your guess is too low.')
    elif guess > secret_number:
        print('Your guess is too high.')
```

These few lines of code check whether the guess is less than or greater than the secret number. In either case, a hint is printed to the screen:

```
    else:
        break  # This condition is the
correct guess!
```

If the guess is neither higher nor lower than the secret number, then it must be equal to the secret number—in which case, you want the program execution to break out of the `for` loop:

```
if guess == secret_number:
    print('Good job! You got it in ' +
str(guesses_taken) + ' guesses!')
else:
```

```
    print('Nope. The number was ' +
str(secret_number))
```

After the `for` loop, the previous `if-else` statement checks whether the player has correctly guessed the number and then prints an appropriate message to the screen. In both cases, the program displays a variable that contains an integer value (`guesses_taken` and `secret_number`). Since it must concatenate these integer values to strings, it passes these variables to the `str()` function, which returns the string value form of these integers. Now these strings can be concatenated with the + operators before finally being passed to the `print()` function call.

A Short Program: Rock, Paper, Scissors

Let's use the programming concepts we've learned so far to create a simple rock, paper, scissors game. The output will look like this:

```
ROCK, PAPER, SCISSORS
0 Wins, 0 Losses, 0 Ties
Enter your move: (r)ock (p)aper (s)cissors or
(q)uit
>p
PAPER versus...
PAPER
It is a tie!
0 Wins, 0 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or
(q)uit
>s
SCISSORS versus...
PAPER
You win!
1 Wins, 0 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or
(q)uit
>q
```

Enter the following source code into the file editor, and save the file as *rpsGame.py*:

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of
wins, losses, and ties.
wins = 0
losses = 0
ties = 0

while True:  # The main game loop
    print('%s Wins, %s Losses, %s Ties' %
(wins, losses, ties))
    while True:  # The player input loop
        print('Enter your move: (r)ock
(p)aper (s)cissors or (q)uit')
        player_move = input('>')
        if player_move == 'q':
            sys.exit()  # Quit the program.
        if player_move == 'r' or player_move
== 'p' or player_move == 's':
            break  # Break out of the player
input loop.
        print('Type one of r, p, s, or q.')

    # Display what the player chose:
    if player_move == 'r':
        print('ROCK versus...')
    elif player_move == 'p':
        print('PAPER versus...')
    elif player_move == 's':
```

```
        print('SCISSORS versus...')

    # Display what the computer chose:
    move_number = random.randint(1, 3)
    if move_number == 1:
        computer_move = 'r'
        print('ROCK')
    elif move_number == 2:
        computer_move = 'p'
        print('PAPER')
    elif move_number == 3:
        computer_move = 's'
        print('SCISSORS')

    # Display and record the win/loss/tie:
    if player_move == computer_move:
        print('It is a tie!')
        ties = ties + 1
    elif player_move == 'r' and computer_move
== 's':
        print('You win!')
        wins = wins + 1
    elif player_move == 'p' and computer_move
== 'r':
        print('You win!')
        wins = wins + 1
    elif player_move == 's' and computer_move
== 'p':
        print('You win!')
        wins = wins + 1
    elif player_move == 'r' and computer_move
== 'p':
        print('You lose!')
        losses = losses + 1
```

```
    elif player_move == 'p' and computer_move
== 's':
        print('You lose!')
        losses = losses + 1
    elif player_move == 's' and computer_move
== 'r':
        print('You lose!')
        losses = losses + 1
```

Let's look at this code line by line, starting at the top:

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of
wins, losses, and ties.
wins = 0
losses = 0
ties = 0
```

First, we import the `random` and `sys` modules so that our program can call the `random.randint()` and `sys.exit()` functions. We also set up three variables to keep track of how many wins, losses, and ties the player has had:

```
while True:  # The main game loop
    print('%s Wins, %s Losses, %s Ties' %
(wins, losses, ties))
    while True:  # The player input loop
        print('Enter your move: (r)ock
(p)aper (s)cissors or (q)uit')
        player_move = input('>')
        if player_move == 'q':
```

```
            sys.exit()  # Quit the program.
        if player_move == 'r' or player_move
== 'p' or player_move == 's':
            break  # Break out of the player
input loop.
        print('Type one of r, p, s, or q.')
```

This program uses a `while` loop inside another `while` loop. The first loop is the main game loop, and a single game of rock, paper, scissors is played on each iteration through this loop. The second loop asks for input from the player, and keeps looping until the player has entered an `r`, `p`, `s`, or `q` for their move. The `r`, `p`, and `s` correspond to rock, paper, and scissors, respectively, while the `q` means the player intends to quit. In that case, `sys.exit()` is called and the program exits. If the player has entered `r`, `p`, or `s`, the execution breaks out of the loop. Otherwise, the program reminds the player to enter `r`, `p`, `s`, or `q` and goes back to the start of the loop:

```
    # Display what the player chose:
    if player_move == 'r':
        print('ROCK versus...')
    elif player_move == 'p':
        print('PAPER versus...')
    elif player_move == 's':
        print('SCISSORS versus...')
```

The player's move is displayed on the screen:

```
    # Display what the computer chose:
    move_number = random.randint(1, 3)
    if move_number == 1:
        computer_move = 'r'
        print('ROCK')
    elif move_number == 2:
        computer_move = 'p'
        print('PAPER')
```

```
    elif move_number == 3:
        computer_move = 's'
        print('SCISSORS')
```

Next, the program randomly selects the computer's move. Because `random.randint()` will always return a random number, the code stores the `1`, `2`, or `3` integer value it returns in a variable named `move_number`. The program then stores an `'r'`, `'p'`, or `'s'` string in `computer_move` based on the integer in `move_number`, as well as displays the computer's move:

```
    # Display and record the win/loss/tie:
    if player_move == computer_move:
        print('It is a tie!')
        ties = ties + 1
    elif player_move == 'r' and computer_move
== 's':
        print('You win!')
        wins = wins + 1
    elif player_move == 'p' and computer_move
== 'r':
        print('You win!')
        wins = wins + 1
    elif player_move == 's' and computer_move
== 'p':
        print('You win!')
        wins = wins + 1
    elif player_move == 'r' and computer_move
== 'p':
        print('You lose!')
        losses = losses + 1
    elif player_move == 'p' and computer_move
== 's':
        print('You lose!')
        losses = losses + 1
```

```
    elif player_move == 's' and computer_move
== 'r':
        print('You lose!')
        losses = losses + 1
```

Finally, the program compares the strings in `player_move` and `computer_move`, and displays the results on the screen. It also increments the `wins`, `losses`, or `ties` variable appropriately. Once the execution reaches the end, it jumps back to the start of the main program loop to begin another game.

If you liked these guess the number and rock, paper, scissors games, you can find the source code to other simple Python programs in *The Big Book of Small Python Projects* (No Starch Press, 2021).

Summary

Loops let your programs execute code over and over again while a certain condition evaluates to `True`. The `break` and `continue` statements are useful if you need to exit a loop or jump back to the loop's start.

These flow control statements will let you write more intelligent programs. You can also use another type of flow control by writing your own functions, which is the topic of the next chapter.

Practice Questions

1. What keys can you press if your Python program is stuck in an infinite loop?
2. What is the difference between `break` and `continue`?
3. What is the difference between `range(10)`, `range(0, 10)`, and `range(0, 10, 1)` in a `for` loop?
4. Write a short program that prints the numbers `1` to `10` using a `for` loop. Then, write an equivalent program that prints the numbers `1` to `10` using a `while` loop.
5. If you had a function named `bacon()` inside a module named `spam`, how would you call it after importing `spam`?