Support the author by purchasing the print/ebook bundle from No Starch Press or separately on Amazon.

Read the author's other Creative Commons licensed Python books.

2
FLOW CONTROL

So, you know the basics of individual instructions and that a program is just a series of instructions. But programming's real strength isn't just running one instruction after another like a weekend errand list. Based on how expressions evaluate, a program can decide to skip instructions, repeat them, or choose one of several instructions to run. In

fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *Flow control statements* can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart, so I'll provide flowchart versions of the code discussed in this chapter. Figure 2-1 shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.
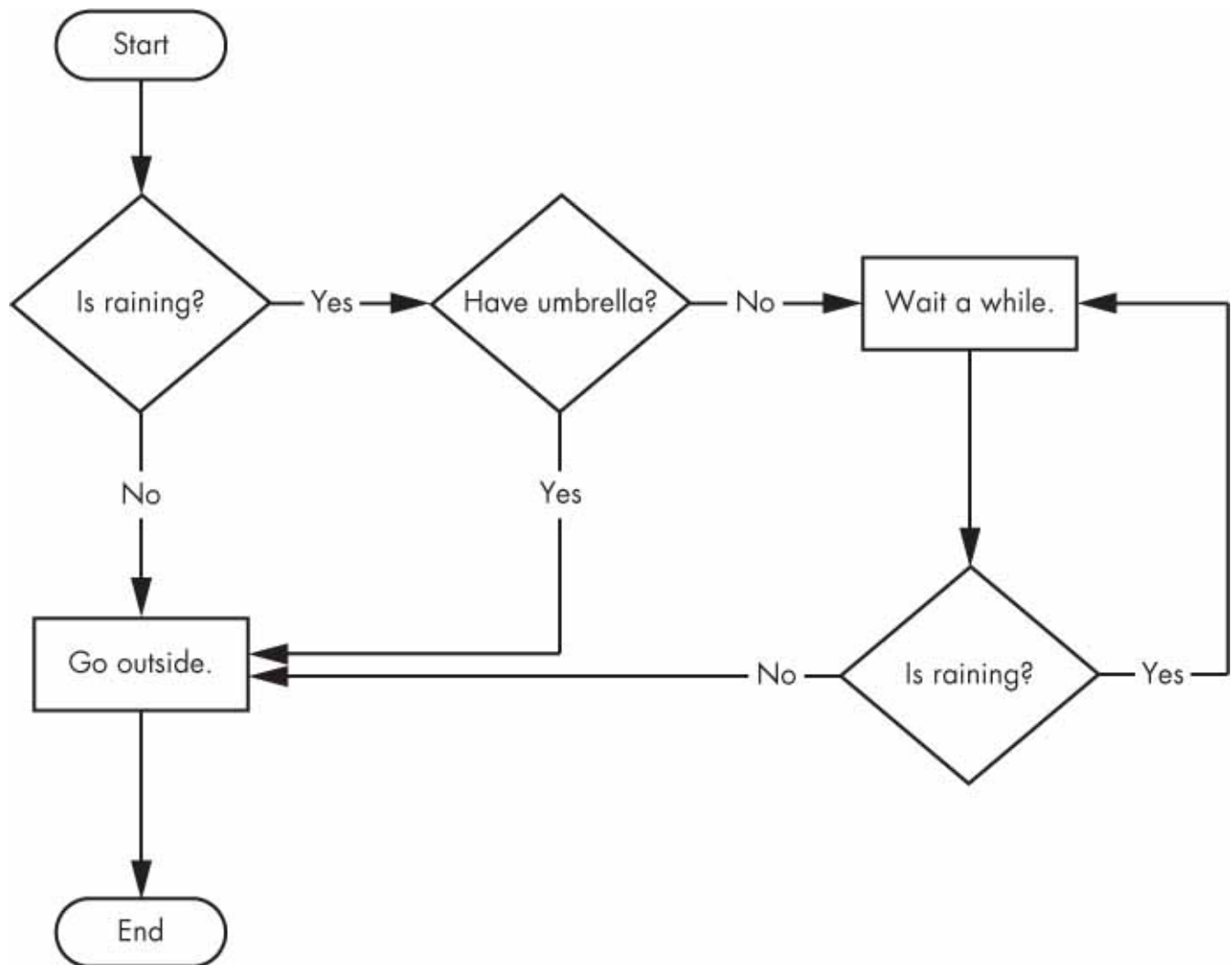

Figure 2-1: A flowchart to tell you what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

But before you learn about flow control statements, you first need to learn how to represent those *yes* and *no* options, and you need to understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

BOOLEAN VALUES

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: True and False. (Boolean is capitalized because the data type is named after mathematician George Boole.) When entered as Python code, the Boolean values True and False lack the quotes you place around strings, and they always start with a capital *T* or *F*, with the rest of the word in lowercase. Enter the following into the interactive shell. (Some of these instructions are intentionally incorrect, and they'll cause error messages to appear.)

---

❶ >>> **spam = True**
  >>> **spam**
  True
❷ >>> **true**
  Traceback (most recent call last):
    File "\<pyshell#2>", line 1, in \<module>
     true
  NameError: name 'true' is not defined
❸ >>> **True = 2 + 2**
  SyntaxError: can't assign to keyword

Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If you don't use the proper case ❷ or you try to use True and False for variable names ❸, Python will give you an error message.

COMPARISON OPERATORS

Comparison operators, also called *relational operators*, compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.

Table 2-1: Comparison Operators

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

These operators evaluate to True or False depending on the values you give them. Let's try some operators now, starting with == and !=.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

As you might expect, == (equal to) evaluates to True when the values on both sides are the same, and != (not equal to) evaluates to True when the two values are different. The == and != operators can actually work with values of any data type.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
❶ >>> 42 == '42'
False
```

Note that an integer or floating-point value will always be unequal to a string value. The expression 42 == '42' ❶ evaluates to False because Python considers the integer 42 to be different from the string '42'.

The <, >, <=, and >= operators, on the other hand, work properly only with integer and floating-point values.

```
>>> 42 < 100
True
>>> 42 > 100
```

False

>>> **42 < 42**

False

>>> **eggCount = 42**

❶ >>> **eggCount <= 42**

True

>>> **myAge = 29**

❷ >>> **myAge >= 10**

True

THE DIFFERENCE BETWEEN THE == AND = OPERATORS

You might have noticed that the == operator (equal to) has two equal signs, while the = operator (assignment) has just one equal sign. It's easy to confuse these two operators with each other. Just remember these points:

- The == operator (equal to) asks whether two values are the same as each other.
- The = operator (assignment) puts the value on the right into the variable on the left.

To help remember which is which, notice that the == operator (equal to) consists of two characters, just like the != operator (not equal to) consists of two characters.

You'll often use comparison operators to compare a variable's value to some other value, like in the eggCount <= 42 ❶ and myAge >= 10 ❷ examples. (After all, instead of entering 'dog' != 'cat' in your code, you could have just entered True.) You'll see more examples of this later when you learn about flow control statements.

BOOLEAN OPERATORS

The three Boolean operators (and, or, and not) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail, starting with the and operator.

Binary Boolean Operators

The and and or operators always take two Boolean values (or expressions), so they're considered *binary* operators. The and operator evaluates an expression to True if *both* Boolean values are True; otherwise, it evaluates to False. Enter some expressions using and into the interactive shell to see it in action.

```
>>> True and True
True
>>> True and False
False
```

A *truth table* shows every possible result of a Boolean operator. Table 2-2 is the truth table for the and operator.

Table 2-2: The and Operator's Truth Table

Expression	Evaluates to . . .
True and True	True
True and False	False
False and True	False
False and False	False

On the other hand, the or operator evaluates an expression to True if *either* of the two Boolean values is True. If both are False, it evaluates to False.

```
>>> False or True
True
>>> False or False
False
```

You can see every possible outcome of the or operator in its truth table, shown in Table 2-3.

Table 2-3: The or Operator's Truth Table

Expression	Evaluates to . . .
True or True	True
True or False	True
False or True	True
False or False	False

The not Operator

Unlike and and or, the not operator operates on only one Boolean value (or expression). This makes it a *unary* operator. The not operator simply evaluates to the opposite

Boolean value.

```
>>> not True
False
❶ >>> not not not not True
True
```

Much like using double negatives in speech and writing, you can nest not operators ❶, though there's never not no reason to do this in real programs. Table 2-4 shows the truth table for not.

Table 2-4: The not Operator's Truth Table

Expression	Evaluates to . . .
not True	False
not False	True

MIXING BOOLEAN AND COMPARISON OPERATORS

Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

Recall that the and, or, and not operators are called Boolean operators because they always operate on the Boolean values True and False. While expressions like 4 < 5 aren't Boolean values, they are expressions that evaluate down to Boolean values. Try entering some Boolean expressions that use comparison operators into the interactive shell.

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer's evaluation process for (4 < 5) and (5 < 6) as the following:

```
(4 < 5) and (5 < 6)
        ↓
 True and (5 < 6)
        ↓
  True and True
        ↓
      True
```
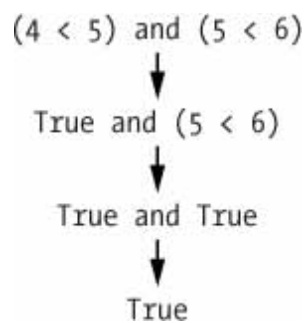
You can also use multiple Boolean operators in an expression, along with the comparison operators:

>>> **2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2**

True

The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the not operators first, then the and operators, and then the or operators.

ELEMENTS OF FLOW CONTROL

Flow control statements often start with a part called the *condition* and are always followed by a block of code called the *clause*. Before you learn about Python's specific flow control statements, I'll cover what a condition and a block are.

Conditions

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; *condition* is just a more specific name in the context of flow control statements. Conditions always evaluate down to a Boolean value, True or False. A flow control statement decides what to do based on whether its condition is True or False, and almost every flow control statement uses a condition.

Blocks of Code

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

- Blocks begin when the indentation increases.
- Blocks can contain other blocks.
- Blocks end when the indentation decreases to zero or to a containing block's indentation.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

```
name = 'Mary'
password = 'swordfish'
if name == 'Mary':
  ❶ print('Hello, Mary')
    if password == 'swordfish':
      ❷ print('Access granted.')
    else:
      ❸ print('Wrong password.')
```

You can view the execution of this program at *https://autbor.com/blocks/*. The first block of code ❶ starts at the line print('Hello, Mary') and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: print('Access Granted.'). The third block ❸ is also one line long: print('Wrong password.').

PROGRAM EXECUTION

In the previous chapter's *hello.py* program, Python started executing instructions at the top of the program going down, one after another. The *program execution* (or simply, *execution*) is a term for the current instruction being executed. If you print the source code on paper and put your finger on each line as it is executed, you can think of your finger as the program execution.

Not all programs execute by simply going straight down, however. If you use your finger to trace through a program with flow control statements, you'll likely find yourself jumping around the source code based on conditions, and you'll probably skip entire clauses.

FLOW CONTROL STATEMENTS

Now, let's explore the most important piece of flow control: the statements themselves. The statements represent the diamonds you saw in the flowchart in Figure 2-1, and they are the actual decisions your programs will make.

if Statements

The most common type of flow control statement is the if statement. An if statement's clause (that is, the block following the if statement) will execute if the statement's condition is True. The clause is skipped if the condition is False.

In plain English, an if statement could be read as, "If this condition is true, execute the code in the clause." In Python, an if statement consists of the following:

- The if 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 if clause)

For example, let's say you have some code that checks to see whether someone's name is Alice. (Pretend name was assigned some value earlier.)

```
if name == 'Alice':
    print('Hi, Alice.')
```

All flow control statements end with a colon and are followed by a new block of code (the clause). This if statement's clause is the block with print('Hi, Alice.'). Figure 2-2 shows what a flowchart of this code would look like.
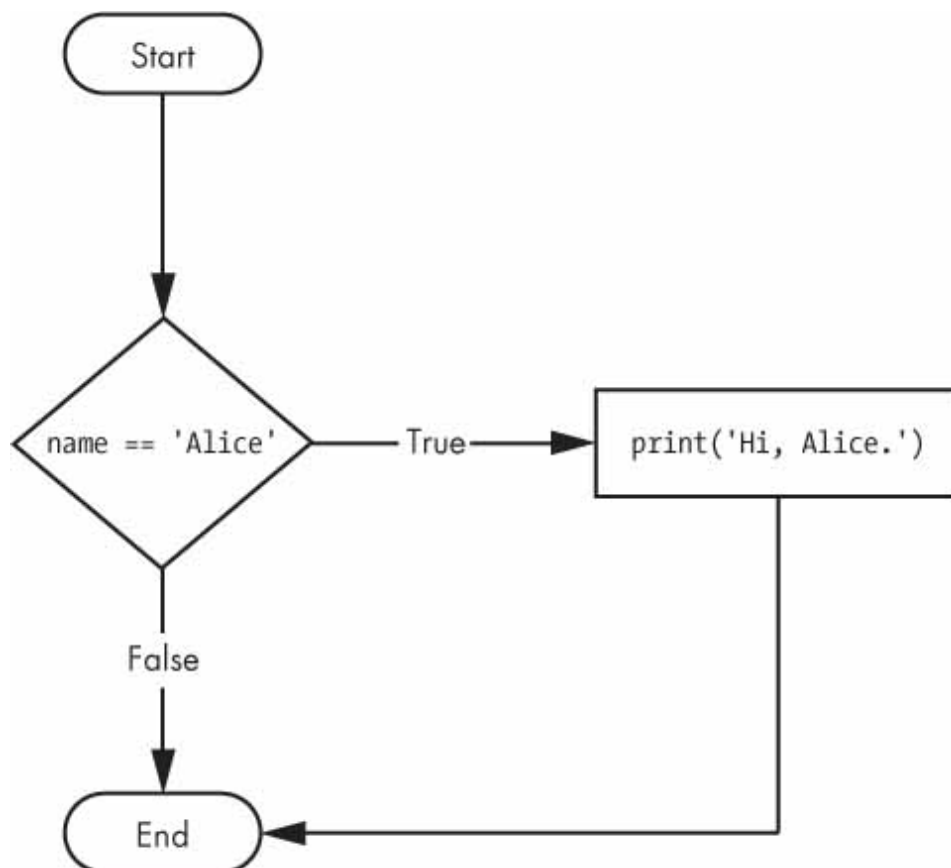
Figure 2-2: The flowchart for an if statement

else Statements

An if clause can optionally be followed by an else statement. The else clause is executed only when the if statement's condition is False. In plain English, an else statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An else

statement doesn't have a condition, and in code, an else statement always consists of the following:

- The else keyword
- A colon
- Starting on the next line, an indented block of code (called the else clause)
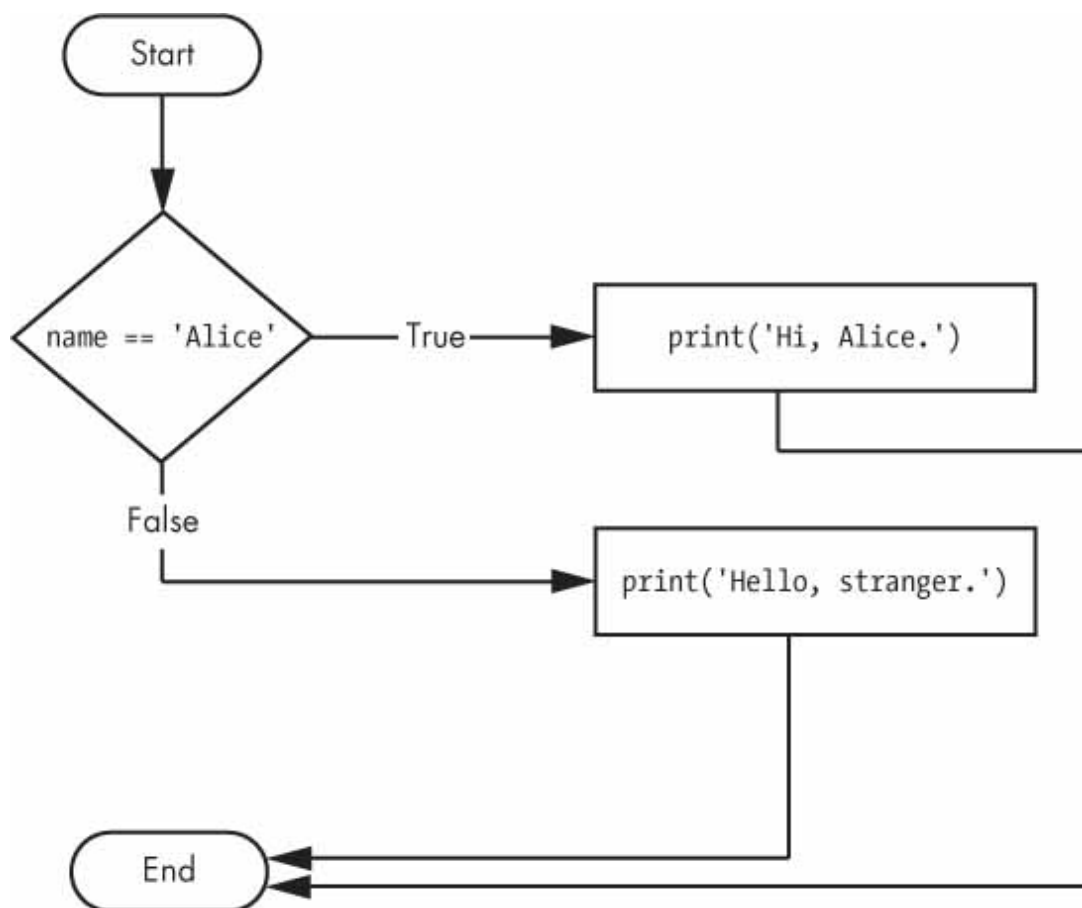
Returning to the Alice example, let's look at some code that uses an else statement to offer a different greeting if the person's name isn't Alice.

```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

Figure 2-3 shows what a flowchart of this code would look like.

Figure 2-3: The flowchart for an else statement

elif Statements

While only one of the if or else clauses will execute, you may have a case where you want one of *many* possible clauses to execute. The elif statement is an "else if" statement that always follows an if or another elif statement. It provides another condition that is

checked only if all of the previous conditions were False. In code, an elif statement always consists of the following:

- The elif 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 elif clause)

Let's add an elif to the name checker to see this statement in action.

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

This time, you check the person's age, and the program will tell them something different if they're younger than 12. You can see the flowchart for this in Figure 2-4.
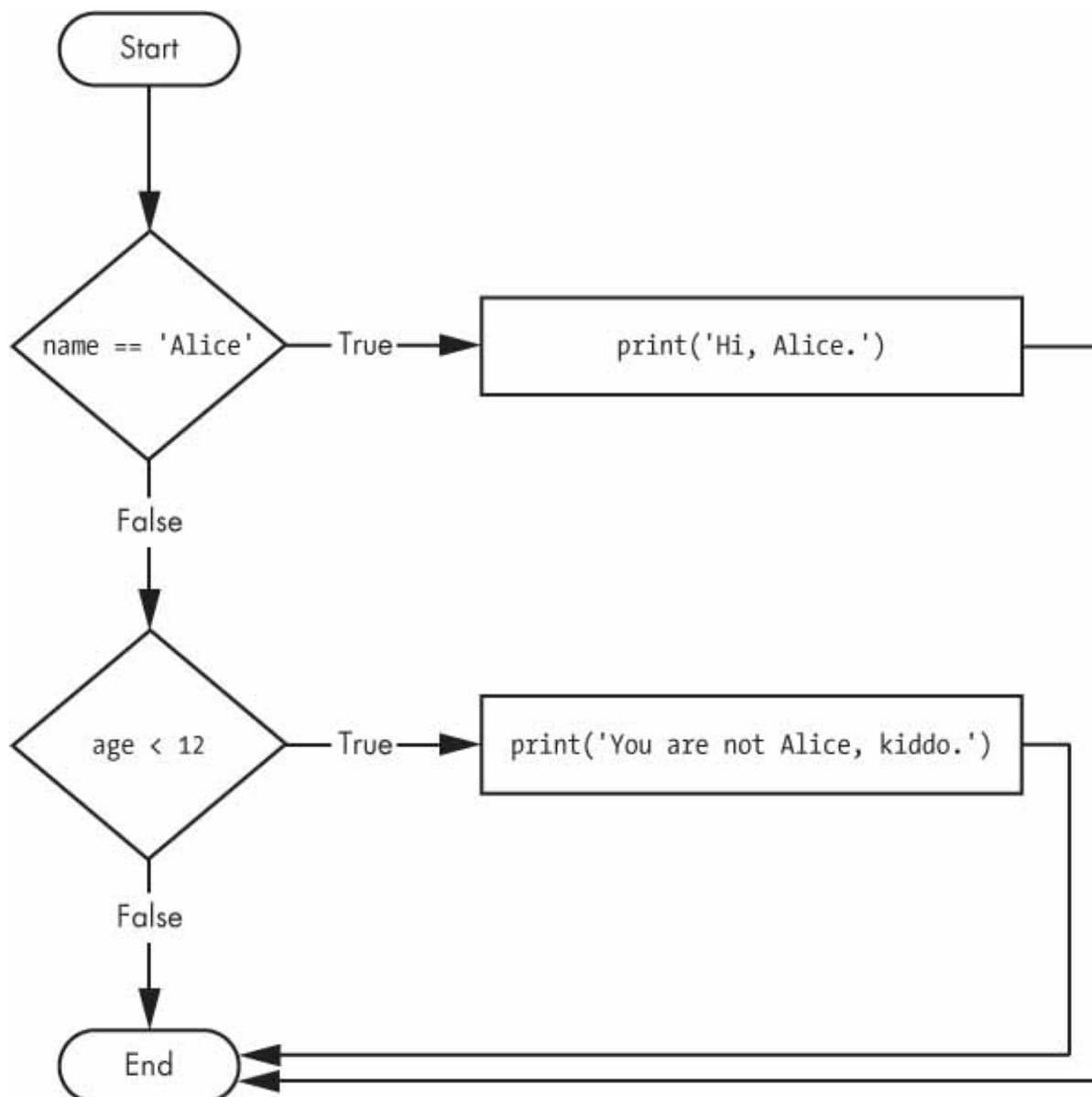
Figure 2-4: The flowchart for an elif statement

The elif clause executes if age < 12 is True and name == 'Alice' is False. However, if both of the conditions are False, then both of the clauses are skipped. It is *not* guaranteed that at least one of the clauses will be executed. When there is a chain of elif statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be True, the rest of the elif clauses are automatically skipped. For example, open a new file editor window and enter the following code, saving it as *vampire.py*:

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

You can view the execution of this program at *https://autbor.com/vampire/*. Here, I've added two more elif statements to make the name checker greet a person with different answers based on age. Figure 2-5 shows the flowchart for this.
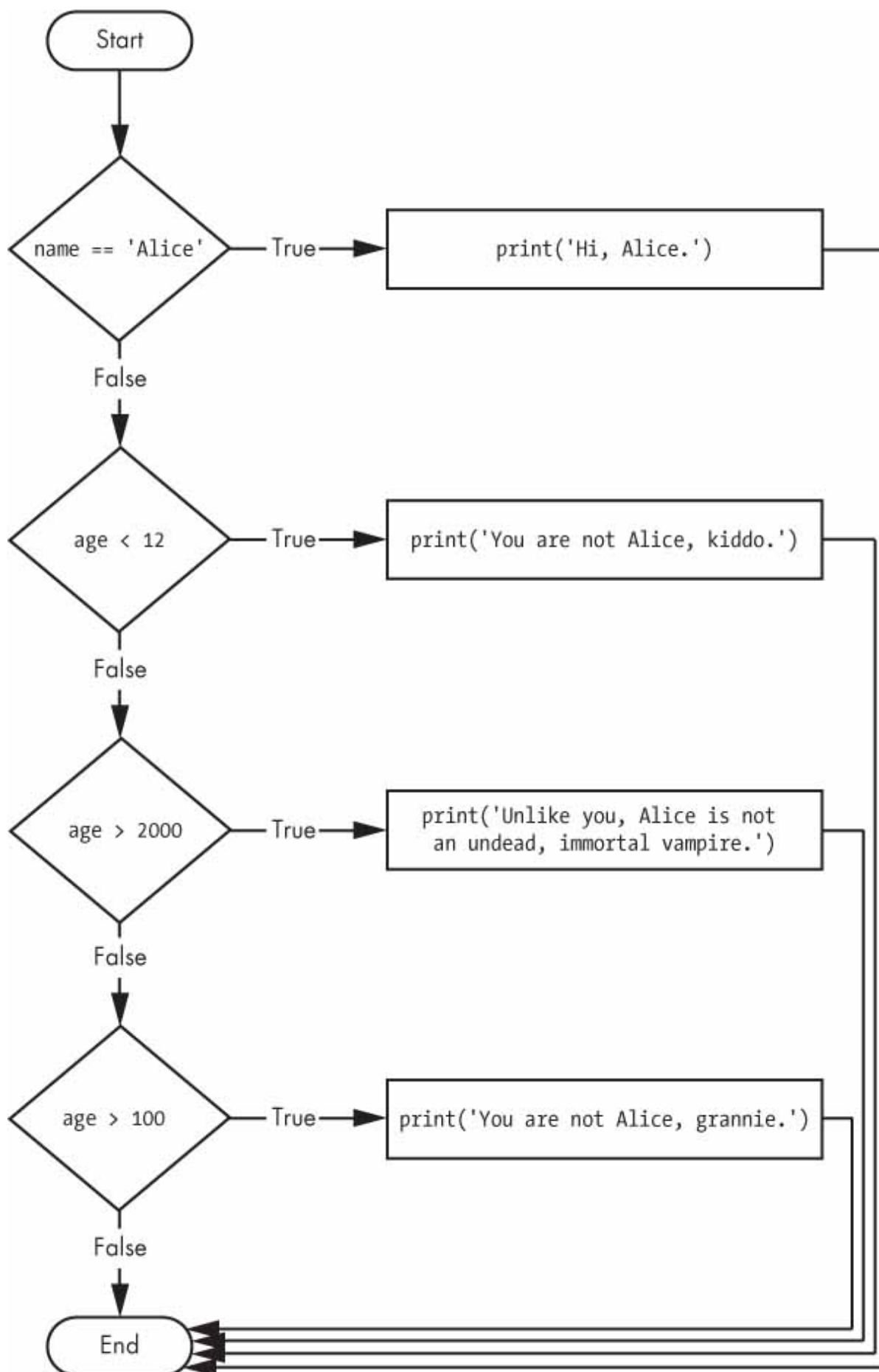
Figure 2-5: The flowchart for multiple elif statements in the vampire.py *program*

The order of the elif statements does matter, however. Let's rearrange them to introduce a bug. Remember that the rest of the elif clauses are automatically skipped once a True condition has been found, so if you swap around some of the clauses in

vampire.py, you run into a problem. Change the code to look like the following, and save it as *vampire2.py*:

```python
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
❶ elif age > 100:
    print('You are not Alice, grannie.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
```

You can view the execution of this program at *https://autbor.com/vampire2/*. Say the age variable contains the value 3000 before this code is executed. You might expect the code to print the string 'Unlike you, Alice is not an undead, immortal vampire.'. However, because the age > 100 condition is True (after all, 3,000 *is* greater than 100) ❶, the string 'You are not Alice, grannie.' is printed, and the rest of the elif statements are automatically skipped. Remember that at most only one of the clauses will be executed, and for elif statements, the order matters!

Figure 2-6 shows the flowchart for the previous code. Notice how the diamonds for age > 100 and age > 2000 are swapped.

Optionally, you can have an else statement after the last elif statement. In that case, it *is* guaranteed that at least one (and only one) of the clauses will be executed. If the conditions in every if and elif statement are False, then the else clause is executed. For example, let's re-create the Alice program to use if, elif, and else clauses.

```python
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

You can view the execution of this program at *https://autbor.com/littlekid/*. Figure 2-7 shows the flowchart for this new code, which we'll save as *littleKid.py*.

In plain English, this type of flow control structure would be "If the first condition is true, do this. Else, if the second condition is true, do that. Otherwise, do something else." When you use if, elif, and else statements together, remember these rules about how to order them to avoid bugs like the one in Figure 2-6. First, there is always exactly one if statement. Any elif statements you need should follow the if statement. Second, if you want to be sure that at least one clause is executed, close the structure with an else statement.
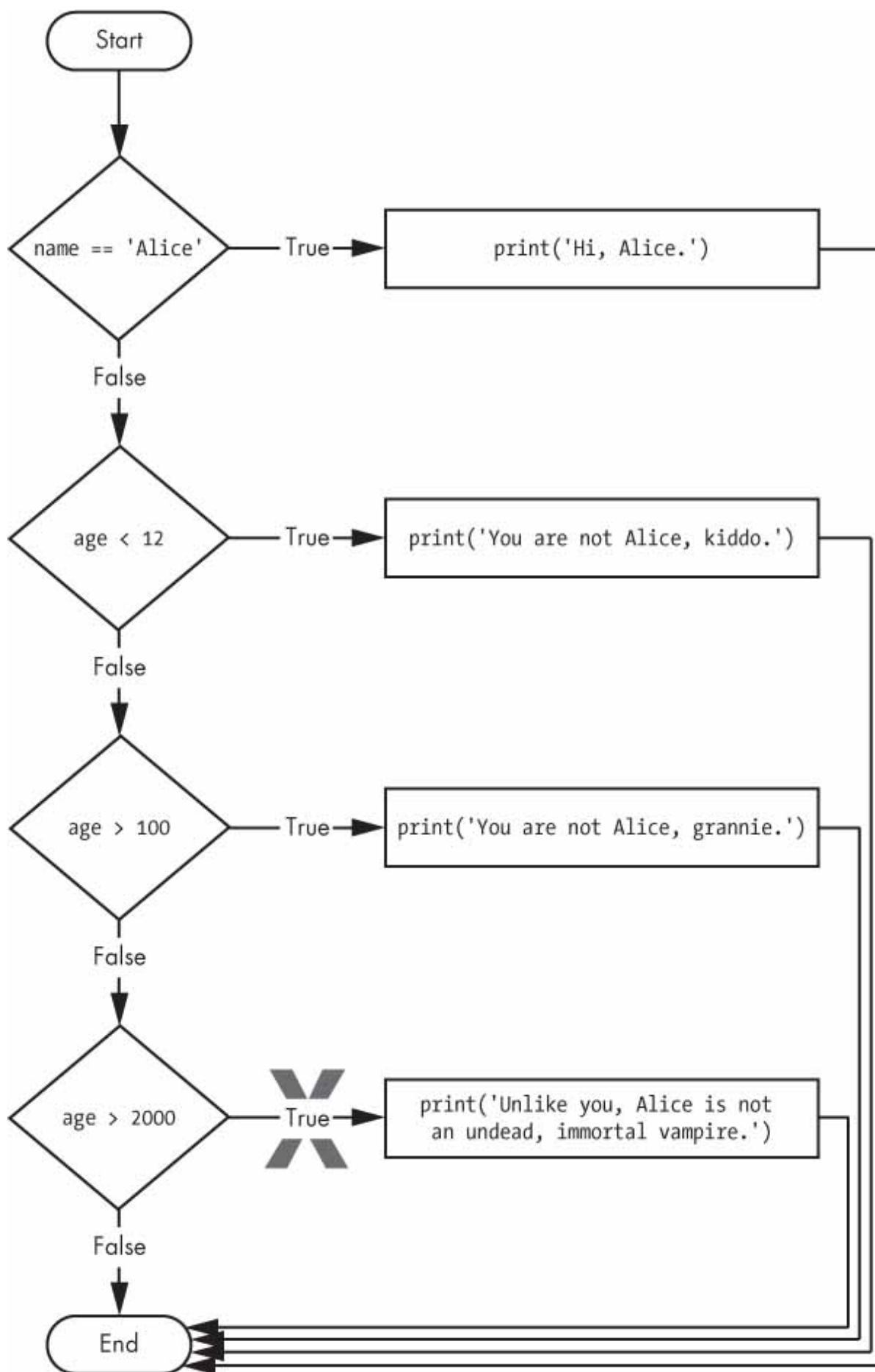
Figure 2-6: The flowchart for the vampire2.py *program. The X path will logically never happen, because if age were greater than 2000, it would have already been greater than 100.*
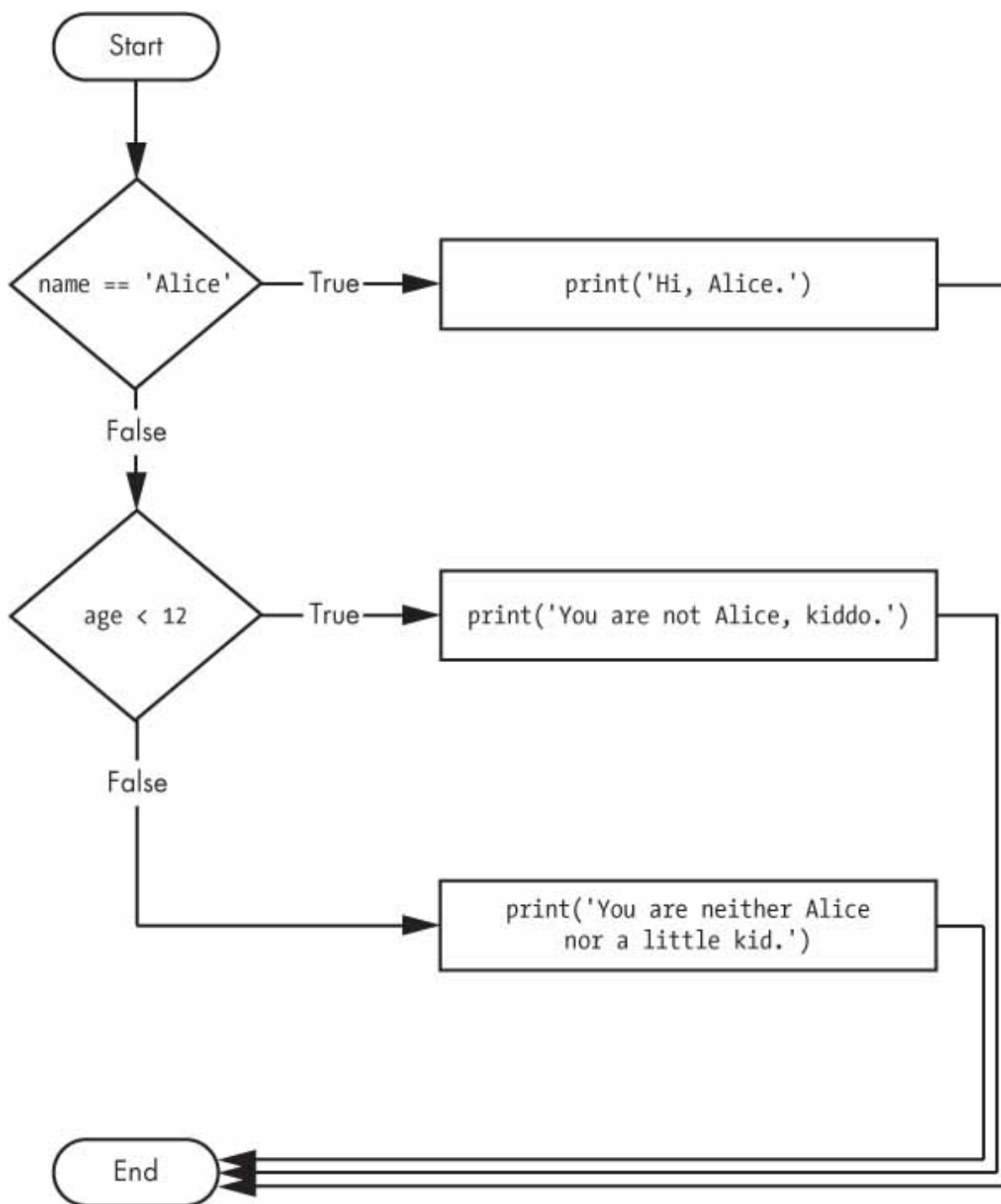
Figure 2-7: Flowchart for the previous littleKid.py *program*

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 while 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)

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 2-8 and 2-9, to see why this happens.
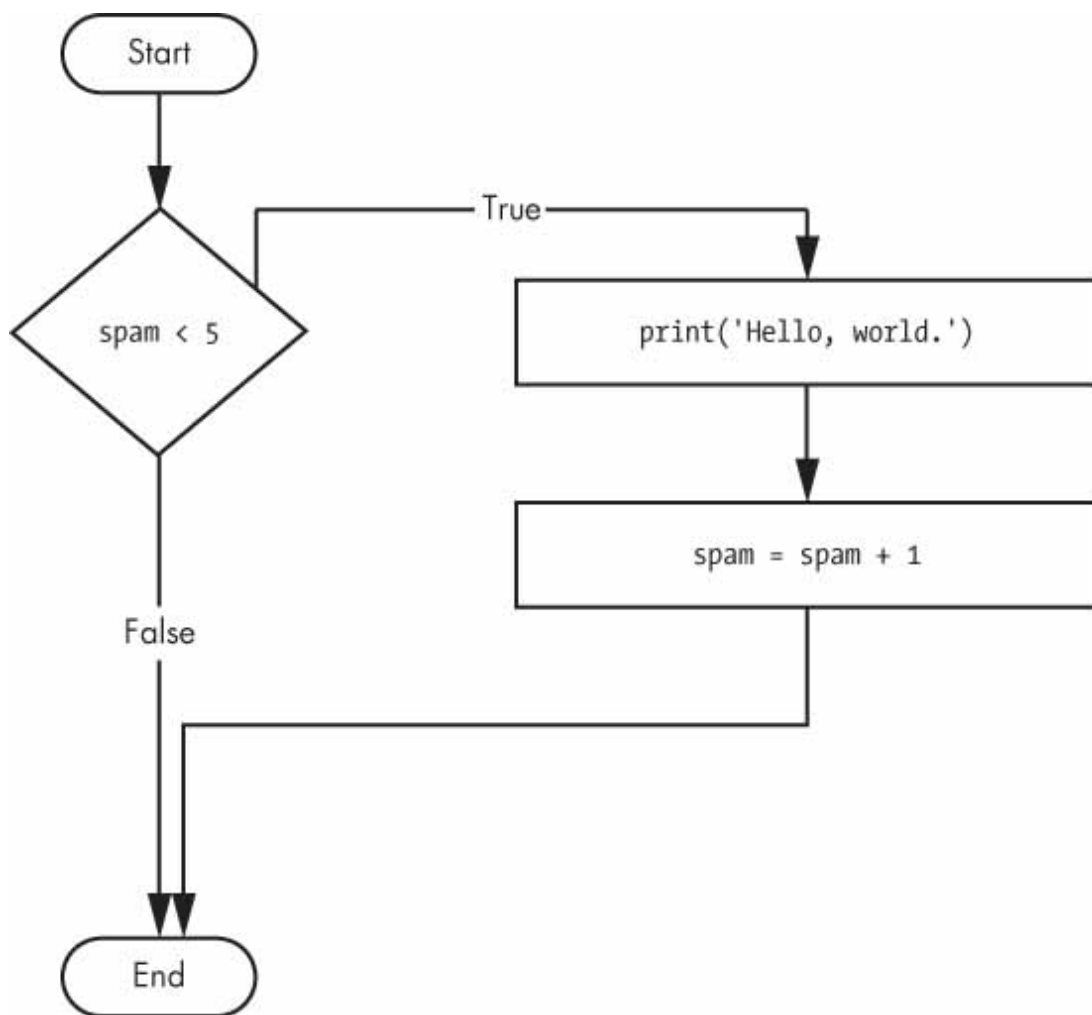
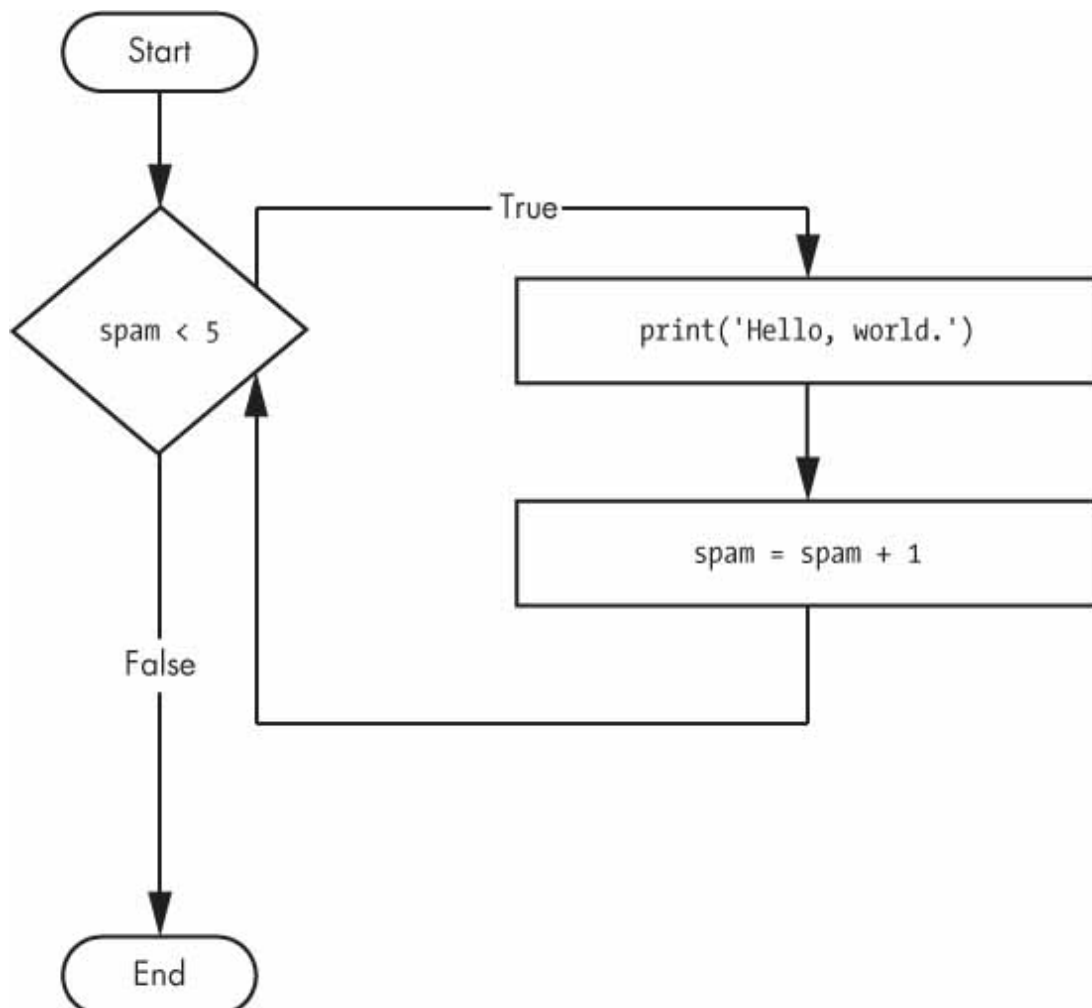*Figure 2-8: The flowchart for the* if *statement code*

Figure 2-9: The flowchart for the while *statement code*

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.

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*:

```
❶ name = ''
❷ while name != 'your name':
      print('Please type your name.')
   ❸ name = input()
❹ print('Thank you!')
```

You can view the execution of this program at *https://autbor.com/yourname/*. First, the program sets the name variable ❶ 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 ❷.

The code inside this clause asks the user to type their name, which is assigned to the name variable ❸. 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', then the condition is True, and the execution enters the while clause again.

But once the user types your name, the condition of the while loop will be 'your name' != 'your name', which evaluates to False. The condition is now False, and instead of the program execution reentering the while loop's clause, Python skips past it and continues running the rest of the program ❹. Figure 2-10 shows a flowchart for the *yourName.py* program.
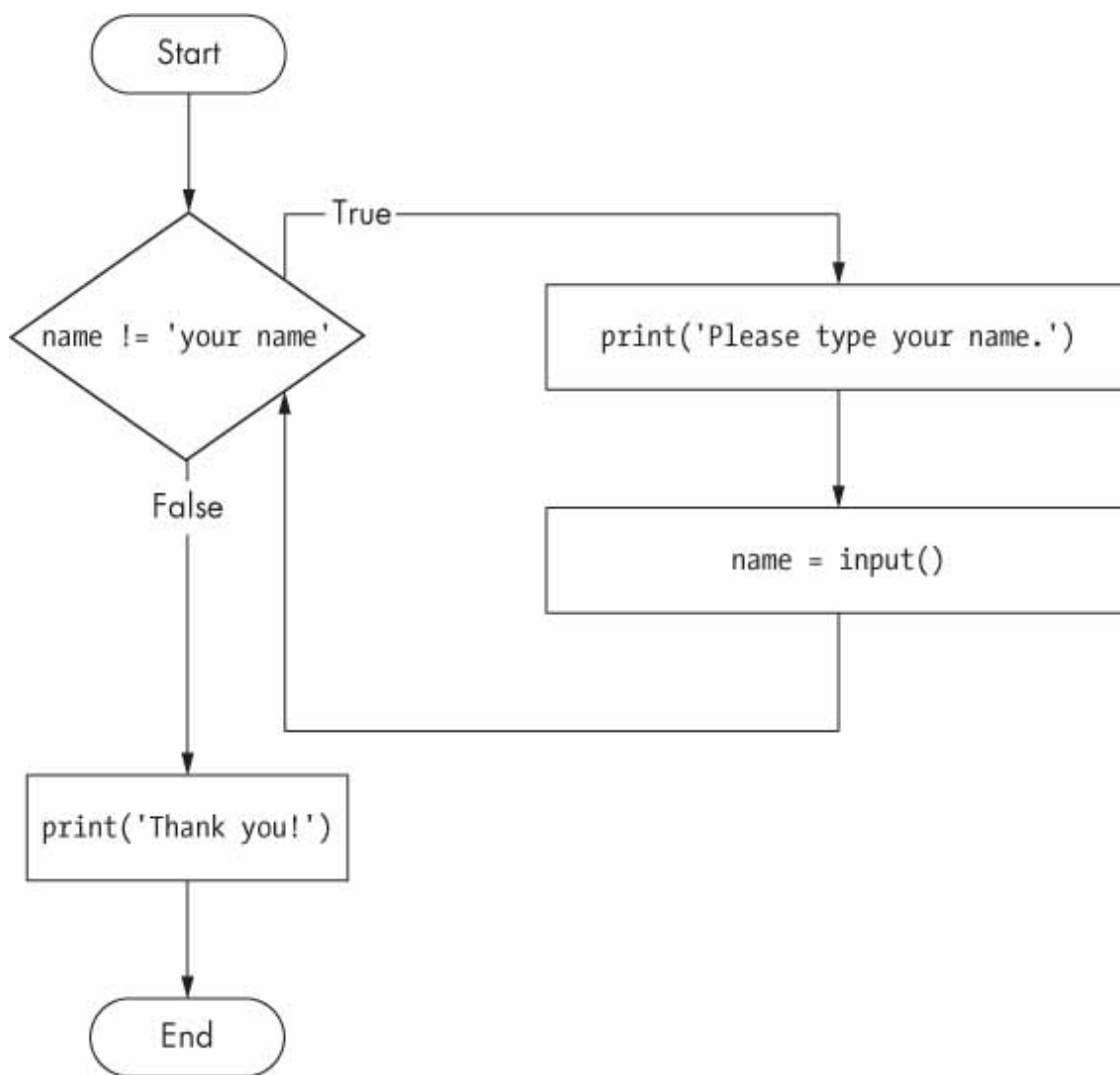
Figure 2-10: A flowchart of the yourName.py *program*

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 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.

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

---

❶ while True:

    print('Please type your name.')

  ❷ name = input()

  ❸ if name == 'your name':

    ❹ break

❺ print('Thank you!')

You can view the execution of this program at *https://autbor.com/yourname2/*. The first line ❶ creates an *infinite loop*; it is a while loop whose condition is always True. (The expression True, after all, always evaluates down 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 ❷. Now, however, while the execution is still inside the while loop, an if statement checks ❸ whether name is equal to 'your name'. If this condition is True, the break statement is run ❹, and the execution moves out of the loop to print('Thank you!') ❺. Otherwise, the if statement's 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 ❶ to recheck the condition. Since this condition is merely the True Boolean value, the execution enters the loop to ask the user to type your name again. See Figure 2-11 for this program's flowchart.

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.
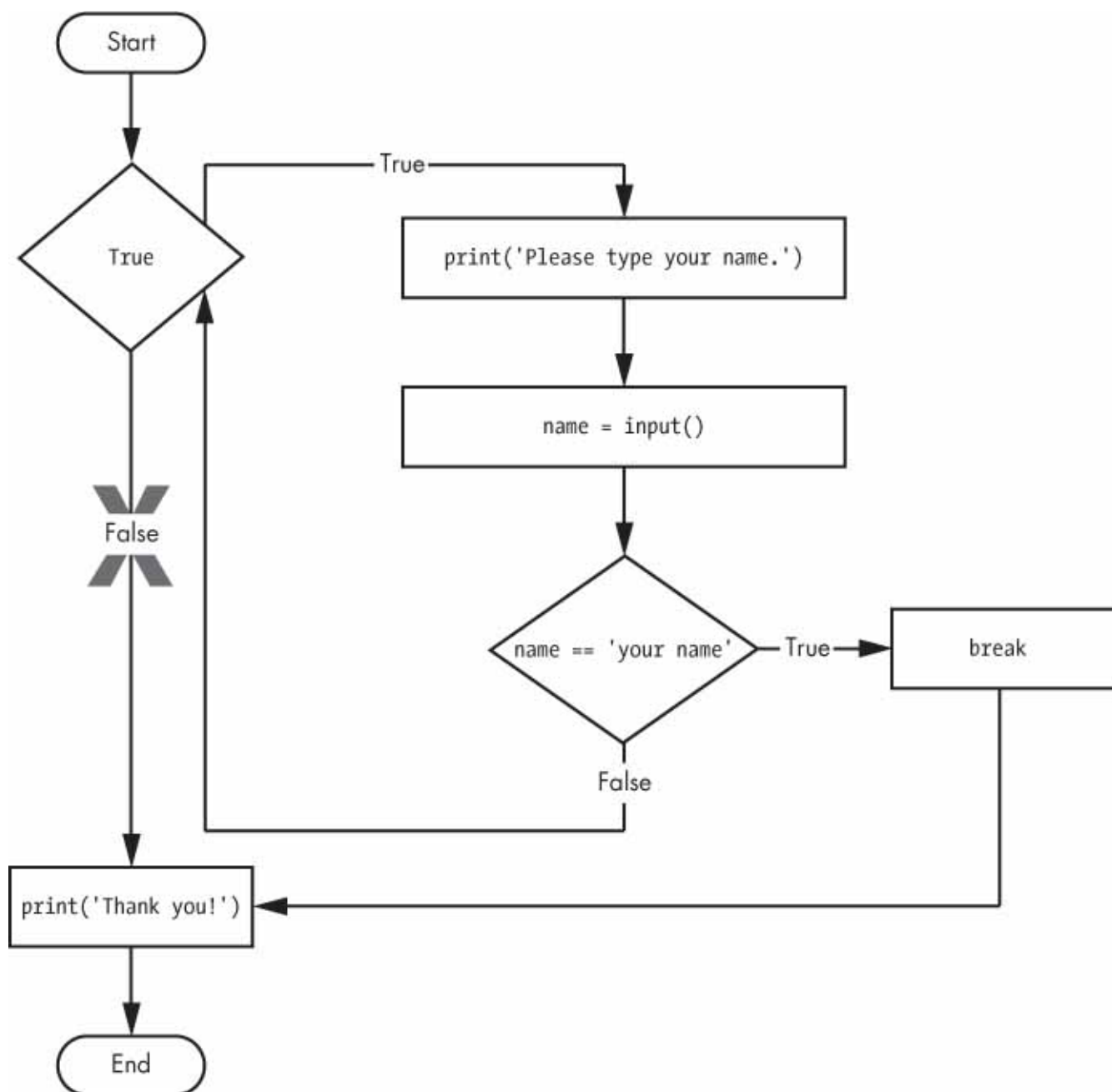
Figure 2-11: The flowchart for the yourName2.py *program with an infinite loop. Note that the X path will logically never happen, because the loop condition is always True.*

continue Statements

Like break statements, continue statements are used 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.)

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*.

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 or select **Shell ▸ Restart Shell** from IDLE's menu. This will send a KeyboardInterrupt error to your

program and cause it to stop immediately. Try stopping a program by creating a simple infinite loop in the file editor, and save the program as *infiniteLoop.py*.

```
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.

```
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, since 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, then 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 2-12 for this program's flowchart.
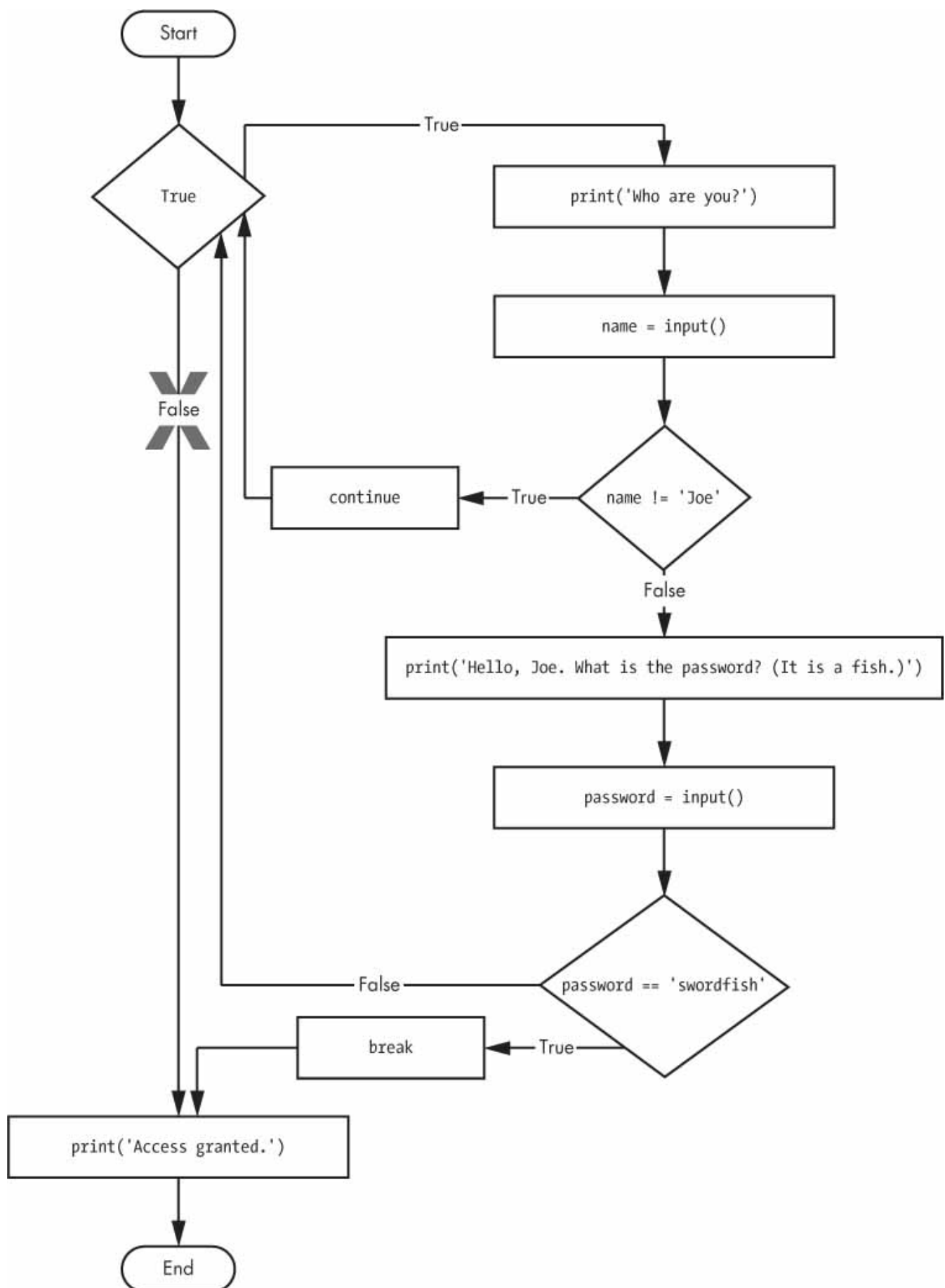
Figure 2-12: A flowchart for swordfish.py. *The X path will logically never happen, because the loop condition is always True.*

"TRUTHY" AND "FALSEY" VALUES

Conditions will consider some values in other data types equivalent to True and 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?')
numOfGuests = int(input())
❷ if numOfGuests:
    ❸ print('Be sure to have enough room for all your guests.')
print('Done')
```

You can view the execution of this program at *https://autbor.com/howmanyguests/*. If the user enters a blank string for name, then the while statement's condition will be True ❶, and the program continues to ask for a name. If the value for numOfGuests is not 0 ❷, then 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 numOfGuests != 0 instead of numOfGuests, but using the truthy and falsey values can make your code easier to read.

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.
```

You can view the execution of this program at *https://autbor.com/hellojoe/*.

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() method with up to three integers passed to it

- A colon

- Starting on the next line, an indented block of code (called the for clause)

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

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

You can view the execution of this program at *https://autbor.com/fivetimesfor/*. 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 Jimmy Five Times (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 2-13 shows a flowchart for the *fiveTimes.py* program.

When you run this program, it should print Jimmy Five Times followed by the value of i five times before leaving the for loop.

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

NOTE

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.
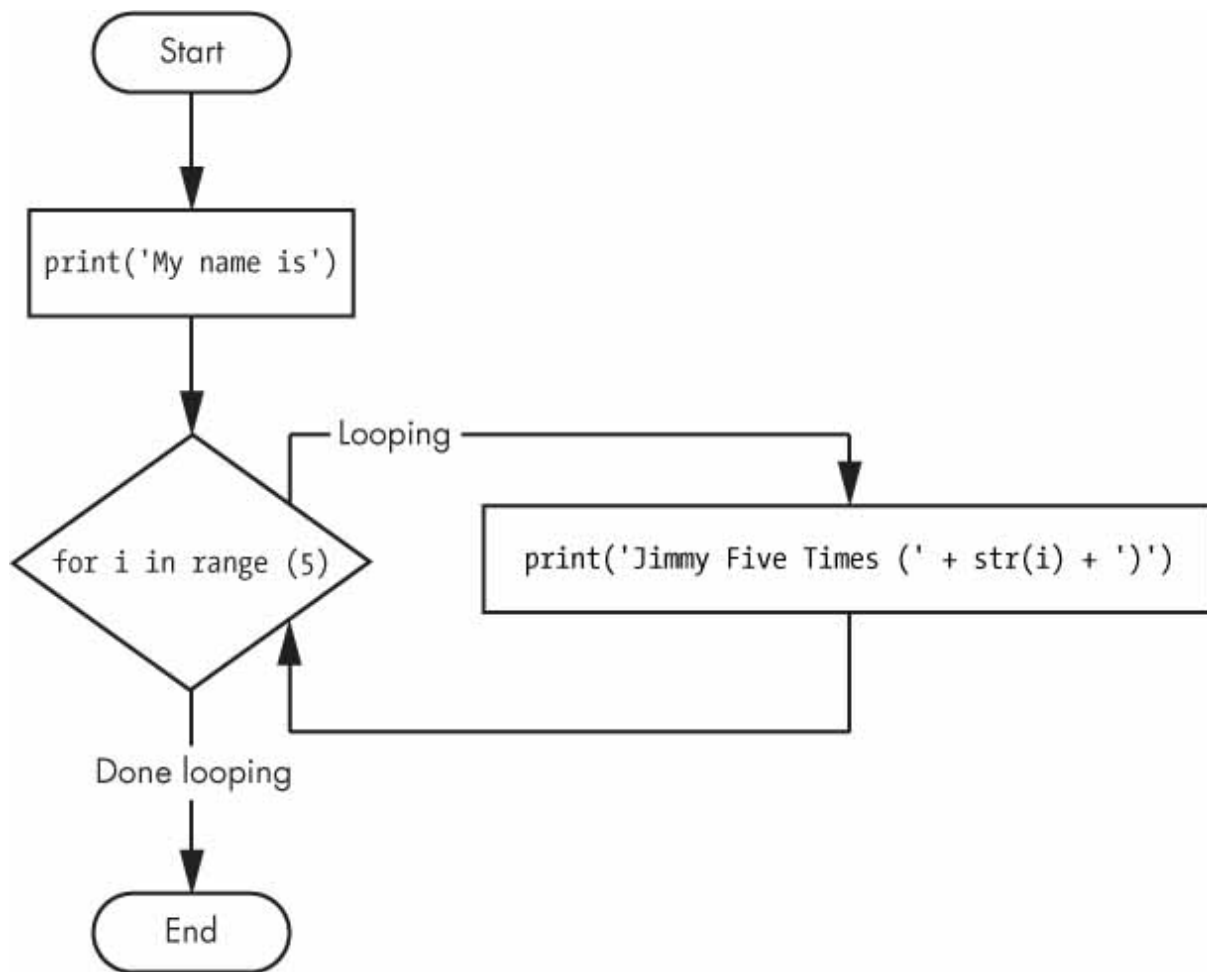

Figure 2-13: The flowchart for fiveTimes.py

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.

---

❶ total = 0
❷ for num in range(101):
   ❸ total = total + num
❹ print(total)

The result should be 5,050. When the program first starts, the total variable is set to 0 ❶. The for loop ❷ then executes total = total + num ❸ 100 times. By the time the loop has finished all of its 100 iterations, every integer from 0 to 100 will have been added to total.

At this point, total is printed to the screen ❹. 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!)

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('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

You can view the execution of this program at *https://autbor.com/fivetimeswhile/*. If you run this program, the output should look the same as the *fiveTimes.py* program, which uses a for loop.

The Starting, Stopping, and Stepping 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 that the variable is increased by 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 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))
```

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 would 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* doesn't have the functions that the real random module has. Don't use the names of any built-in Python functions either, such as print() or input().

Problems like these are uncommon, but can be tricky to solve. As you gain more programming experience, you'll become more aware of the standard names used by Python's modules and functions, and will run into these problems less frequently.

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

```
4
1
8
4
1
```

You can view the execution of this program at *https://autbor.com/printrandom/*. The random.randint() function call evaluates to a random integer value between the two integers that you pass it. Since randint() is in the random module, you must first type **random.** in front of the function name to tell Python to look for this function inside the random module.

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. We'll learn more about them later in the book.

from import Statements

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 will not 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 the sys.exit() Function

The last flow control concept to cover is how to terminate the program. Programs always terminate if the program execution reaches the bottom of the instructions. However, you can cause the program to terminate, or exit, 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 IDLE. 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 let's 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 guessed my number in 4 guesses!

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

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

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

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

if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + '
guesses!')
```

```
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

You can view the execution of this program at *https://autbor.com/guessthenumber/*. Let's look at this code line by line, starting at the top.

```
# This is a guess the number game.
import random
secretNumber = 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 secretNumber.

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

```
# Ask the player to guess 6 times.
for guessesTaken 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 that lets the player enter a guess and checks that guess is 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. Since 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 < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
```

These few lines of code check to see 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.

```python
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

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 (guessesTaken and secretNumber). 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, 1 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
s
SCISSORS versus...
PAPER
You win!
1 Wins, 1 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
q
```

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

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

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

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

```
    # Display and record the win/loss/tie:
    if playerMove == computerMove:
        print('It is a tie!')
        ties = ties + 1
    elif playerMove == 'r' and computerMove == 's':
        print('You win!')
        wins = wins + 1
    elif playerMove == 'p' and computerMove == 'r':
        print('You win!')
        wins = wins + 1
    elif playerMove == 's' and computerMove == 'p':
        print('You win!')
        wins = wins + 1
    elif playerMove == 'r' and computerMove == 'p':
        print('You lose!')
        losses = losses + 1
    elif playerMove == 'p' and computerMove == 's':
        print('You lose!')
        losses = losses + 1
    elif playerMove == 's' and computerMove == '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` module 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')
        playerMove = input()
        if playerMove == 'q':
            sys.exit() # Quit the program.
        if playerMove == 'r' or playerMove == 'p' or playerMove == '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 of another while loop. The first loop is the main game loop, and a single game of rock, paper, scissors is player 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 playerMove == 'r':
        print('ROCK versus...')
    elif playerMove == 'p':
        print('PAPER versus...')
    elif playerMove == 's':
        print('SCISSORS versus...')
```

The player's move is displayed on the screen.

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

```
    computerMove = 's'
    print('SCISSORS')
```

Next, the computer's move is randomly selected. Since random.randint() can only return a random number, the 1, 2, or 3 integer value it returns is stored in a variable named randomNumber. The program stores a 'r', 'p', or 's' string in computerMove based on the integer in randomNumber, as well as displays the computer's move.

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

Finally, the program compares the strings in playerMove and computerMove, 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.

SUMMARY

By using expressions that evaluate to True or False (also called conditions), you can write programs that make decisions on what code to execute and what code to skip. You can

also execute code over and over again in a loop 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 are the two values of the Boolean data type? How do you write them?

2. What are the three Boolean operators?

3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).

4. What do the following expressions evaluate to?

(5 > 4) and (3 == 5)

not (5 > 4)

(5 > 4) or (3 == 5)

not ((5 > 4) or (3 == 5))

(True and True) and (True == False)

(not False) or (not True)

---

5. What are the six comparison operators?

6. What is the difference between the equal to operator and the assignment operator?

7. Explain what a condition is and where you would use one.

8. Identify the three blocks in this code:

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
    else:
        print('ham')
    print('spam')
print('spam')
```

---

9. Write code that prints Hello if 1 is stored in spam, prints Howdy if 2 is stored in spam, and prints Greetings! if anything else is stored in spam.

10. What keys can you press if your program is stuck in an infinite loop?

11. What is the difference between break and continue?

12. What is the difference between range(10), range(0, 10), and range(0, 10, 1) in a for loop?

13. 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.

14. If you had a function named bacon() inside a module named spam, how would you call it after importing spam?

Extra credit: Look up the round() and abs() functions on the internet, and find out what they do. Experiment with them in the interactive shell.