CHAPTER

4 Repetition Structures

TOPICS

4.1 Introduction to Repetition Structures

CONCEPT: A repetition structure causes a statement or set of statements to execute repeatedly.

Programmers commonly have to write code that performs the same task over and over. For example, suppose you have been asked to write a program that calculates a 10 percent sales commission for several salespeople. Although it would not be a good design, one approach would be to write the code to calculate one salesperson's commission, and then repeat that code for each salesperson. For example, look at the following:

```
# Get a salesperson's sales and commission rate.
sales = float(input('Enter the amount of sales: '))
comm_rate = float(input('Enter the commission rate: '))

# Calculate the commission.
commission = sales * comm_rate

# Display the commission.
print('The commission is $', format(commission, ',.2f'), sep='')

# Get another salesperson's sales and commission rate.
sales = float(input('Enter the amount of sales: '))
comm_rate = float(input('Enter the commission rate: '))

# Calculate the commission.
commission = sales * comm_rate
```

```
# Display the commission.
print('The commission is $', format(commission, ',.2f'), sep='')

# Get another salesperson's sales and commission rate.
sales = float(input('Enter the amount of sales: '))
comm_rate = float(input('Enter the commission rate: '))

# Calculate the commission.
commission = sales * comm_rate
# Display the commission.

print('The commission is $', format(commission, ',.2f'), sep='')
```

And this code goes on and on . . .

As you can see, this code is one long sequence structure containing a lot of duplicated code. There are several disadvantages to this approach, including the following:

- The duplicated code makes the program large.
- Writing a long sequence of statements can be time consuming.
- If part of the duplicated code has to be corrected or changed, then the correction or change has to be done many times.

Instead of writing the same sequence of statements over and over, a better way to repeatedly perform an operation is to write the code for the operation once, then place that code in a structure that makes the computer repeat it as many times as necessary. This can be done with a *repetition structure,* which is more commonly known as a *loop*.

Condition-Controlled and Count-Controlled Loops

In this chapter, we will look at two broad categories of loops: condition-controlled and count-controlled. A *condition-controlled loop* uses a true/false condition to control the number of times that it repeats. A *count-controlled loop* repeats a specific number of times. In Python, you use the `while` statement to write a condition-controlled loop, and you use the `for` statement to write a count-controlled loop. In this chapter, we will demonstrate how to write both types of loops.

Checkpoint

4.1 What is a repetition structure?

4.2 What is a condition-controlled loop?

4.3 What is a count-controlled loop?

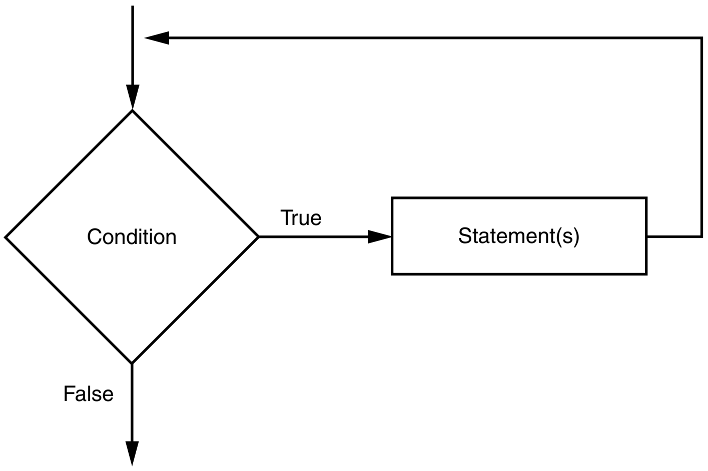
4.2 The `while` Loop: A Condition-Controlled Loop

VideoNote
The `while` Loop

CONCEPT: A condition-controlled loop causes a statement or set of statements to repeat as long as a condition is true. In Python, you use the `while` statement to write a condition-controlled loop.

The `while` loop gets its name from the way it works: *while a condition is true, do some task*. The loop has two parts: (1) a condition that is tested for a true or false value, and (2) a statement or set of statements that is repeated as long as the condition is true. Figure 4-1 shows the logic of a `while` loop.

Figure 4-1 The logic of a `while` loop



The diamond symbol represents the condition that is tested. Notice what happens if the condition is true: one or more statements are executed, and the program's execution flows back to the point just above the diamond symbol. The condition is tested again, and if it is true, the process repeats. If the condition is false, the program exits the loop. In a flowchart, you will always recognize a loop when you see a flow line going back to a previous part of the flowchart.

Here is the general format of the `while` loop in Python:

```
while condition:
      statement
      statement
      etc.
```

For simplicity, we will refer to the first line as the ***while** clause*. The `while` clause begins with the word `while`, followed by a Boolean *`condition`* that will be evaluated as either true or false. A colon appears after the *`condition`*. Beginning at the next line is a block of statements. (Recall from Chapter 3 that all of the statements in a block must be consistently indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.)

When the `while` loop executes, the *`condition`* is tested. If the *`condition`* is true, the statements that appear in the block following the `while` clause are executed, and the loop starts over. If the *`condition`* is false, the program exits the loop. Program 4-1 shows how we might use a `while` loop to write the commission calculating program that was described at the beginning of this chapter.

Program 4-1 (commission.py)

```
 1  # This program calculates sales commissions.
 2
 3  # Create a variable to control the loop.
 4  keep_going = 'y'
 5
 6  # Calculate a series of commissions.
 7  while keep_going == 'y':
 8      # Get a salesperson's sales and commission rate.
 9      sales = float(input('Enter the amount of sales: '))
10      comm_rate = float(input('Enter the commission rate: '))
11
12      # Calculate the commission.
13      commission = sales * comm_rate
14
15      # Display the commission.
16      print('The commission is $',
17            format(commission, ',.2f'), sep='')
18
19      # See if the user wants to do another one.
20      keep_going = input('Do you want to calculate another ' +
21                         'commission (Enter y for yes): ')
```

Program Output (with input shown in bold)

```
Enter the amount of sales: 10000.00 Enter
Enter the commission rate: 0.10 Enter
The commission is $1,000.00
Do you want to calculate another commission (Enter y for yes): y Enter
Enter the amount of sales: 20000.00 Enter
Enter the commission rate: 0.15 Enter
The commission is $3,000.00
Do you want to calculate another commission (Enter y for yes): y Enter
Enter the amount of sales: 12000.00 Enter
Enter the commission rate: 0.10 Enter
The commission is $1,200.00
Do you want to calculate another commission (Enter y for yes): n Enter
```

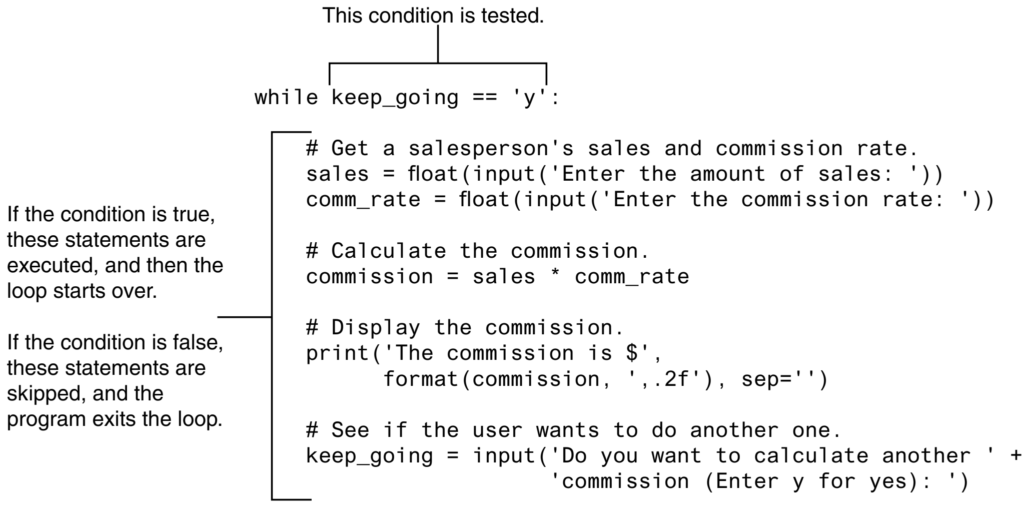
In line 4, we use an assignment statement to create a variable named `keep_going`. Notice the variable is assigned the value `'y'`. This initialization value is important, and in a moment you will see why.

Line 7 is the beginning of a `while` loop, which starts like this:

```
while keep_going == 'y':
```

Notice the condition that is being tested: `keep_going =='y'`. The loop tests this condition, and if it is true, the statements in lines 8 through 21 are executed. Then, the loop starts over at line 7. It tests the expression `keep_going =='y'` and if it is true, the statements in lines 8 through 21 are executed again. This cycle repeats until the expression `keep_going =='y'` is tested in line 7 and found to be false. When that happens, the program exits the loop. This is illustrated in Figure 4-2.

Figure 4-2 The `while` loop


This condition is tested.
If the condition is true, these statements are executed, and then the loop starts over.
If the condition is false, these statements are skipped, and the program exits the loop.

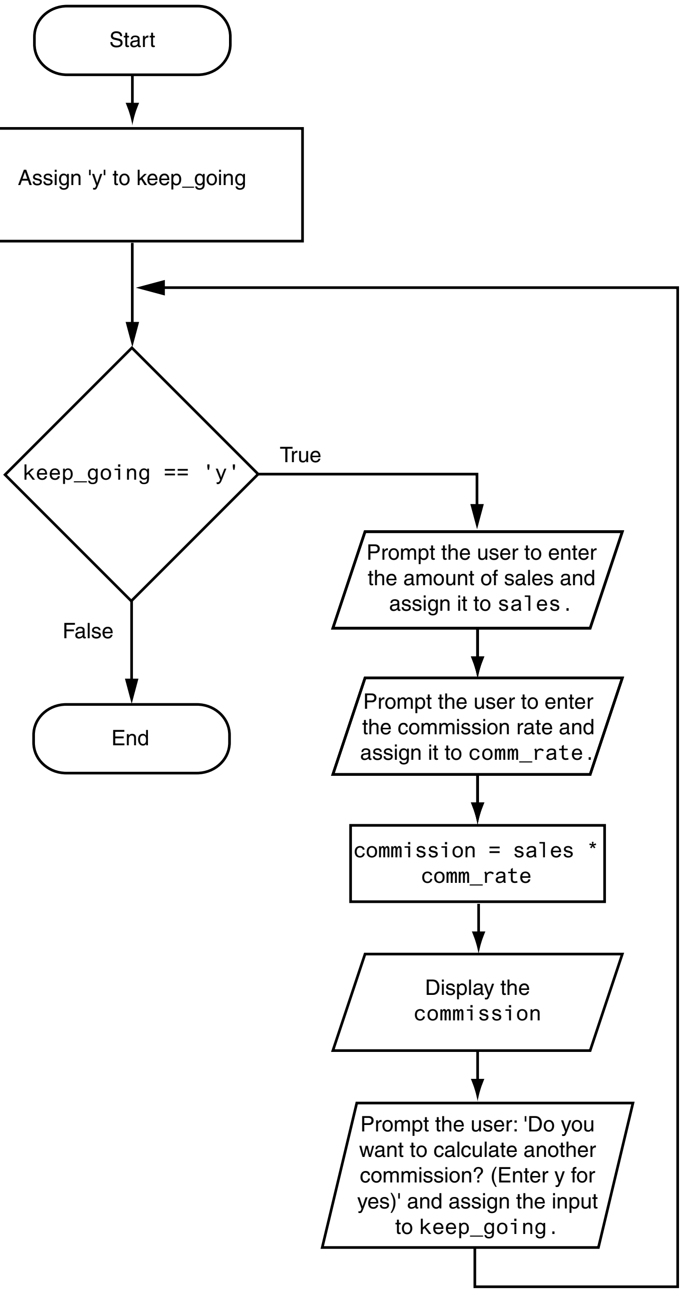
```
while keep_going == 'y':

    # Get a salesperson's sales and commission rate.
    sales = float(input('Enter the amount of sales: '))
    comm_rate = float(input('Enter the commission rate: '))

    # Calculate the commission.
    commission = sales * comm_rate

    # Display the commission.
    print('The commission is $',
          format(commission, ',.2f'), sep='')

    # See if the user wants to do another one.
    keep_going = input('Do you want to calculate another ' +
                       'commission (Enter y for yes): ')
```

In order for this loop to stop executing, something has to happen inside the loop to make the expression `keep_going == 'y'` false. The statement in lines 20 through 21 take care of this. This statement displays the prompt "Do you want to calculate another commission (Enter y for yes)." The value that is read from the keyboard is assigned to the `keep_going` variable. If the user enters y (and it must be a lowercase y), then the expression `keep_going == 'y'` will be true when the loop starts over. This will cause the statements in the body of the loop to execute again. But if the user enters anything other than lowercase y, the expression will be false when the loop starts over, and the program will exit the loop.

Now that you have examined the code, look at the program output in the sample run. First, the user entered 10000.00 for the sales and 0.10 for the commission rate. Then, the program displayed the commission for that amount, which is $1,000.00. Next the user is prompted "Do you want to calculate another commission? (Enter y for yes)." The user entered y, and the loop started the steps over. In the sample run, the user went through this process three times. Each execution of the body of a loop is known as an *iteration*. In the sample run, the loop iterated three times.

Figure 4-3 shows a flowchart for the `main` function. In the flowchart, we have a repetition structure, which is the `while` loop. The condition `keep_going =='y'` is tested, and if it is true, a series of statements are executed and the flow of execution returns to the point just above the conditional test.

Figure 4-3 Flowchart for Program 4-1


Start
Assign 'y' to keep_going
keep_going == 'y'
True
False
End
Prompt the user to enter the amount of sales and assign it to sales.
Prompt the user to enter the commission rate and assign it to comm_rate.
commission = sales * comm_rate
Display the commission
Prompt the user: 'Do you want to calculate another commission? (Enter y for yes)' and assign the input to keep_going.

The `while` Loop Is a Pretest Loop

The `while` loop is known as a *pretest* loop, which means it tests its condition *before* performing an iteration. Because the test is done at the beginning of the loop, you usually have to perform some steps prior to the loop to make sure that the loop executes at least once. For example, the loop in Program 4-1 starts like this:

```
while keep_going == 'y':
```

The loop will perform an iteration only if the expression `keep_going =='y'` is true. This means that (a) the `keep_going` variable has to exist, and (b) it has to reference the value `'y'`. To make sure the expression is true the first time that the loop executes, we assigned the value `'y'` to the `keep_going` variable in line 4 as follows:

```
keep_going = 'y'
```

By performing this step we know that the condition `keep_going =='y'` will be true the first time the loop executes. This is an important characteristic of the while loop: it will never execute if its condition is false to start with. In some programs, this is exactly what you want. The following *In the Spotlight* section gives an example.

In the Spotlight:

Designing a Program with a `while` Loop

A project currently underway at Chemical Labs, Inc. requires that a substance be continually heated in a vat. A technician must check the substance's temperature every 15 minutes. If the substance's temperature does not exceed 102.5 degrees Celsius, then the technician does nothing. However, if the temperature is greater than 102.5 degrees Celsius, the technician must turn down the vat's thermostat, wait 5 minutes, and check the temperature again. The technician repeats these steps until the temperature does not exceed 102.5 degrees Celsius. The director of engineering has asked you to write a program that guides the technician through this process.

Here is the algorithm:

1. Get the substance's temperature.
2. Repeat the following steps as long as the temperature is greater than 102.5 degrees Celsius:
   a. Tell the technician to turn down the thermostat, wait 5 minutes, and check the temperature again.
   b. Get the substance's temperature.
3. After the loop finishes, tell the technician that the temperature is acceptable and to check it again in 15 minutes.

After reviewing this algorithm, you realize that steps 2(a) and 2(b) should not be performed if the test condition (temperature is greater than 102.5) is false to begin with. The `while` loop will work well in this situation, because it will not execute even once if its condition is false. Program 4-2 shows the code for the program.

Program 4-2 (temperature.py)

```python
# This program assists a technician in the process
# of checking a substance's temperature.

# Named constant to represent the maximum
# temperature.
MAX_TEMP = 102.5

# Get the substance's temperature.
temperature = float(input("Enter the substance's Celsius temperature: "))

# As long as necessary, instruct the user to
# adjust the thermostat.
while temperature > MAX_TEMP:
    print('The temperature is too high.')
    print('Turn the thermostat down and wait')
    print('5 minutes. Then take the temperature')
    print('again and enter it.')
    temperature = float(input('Enter the new Celsius temperature: '))

# Remind the user to check the temperature again
# in 15 minutes.
print('The temperature is acceptable.')
print('Check it again in 15 minutes.')
```

Program Output (with input shown in bold)

```
Enter the substance's Celsius temperature: 104.7 Enter
The temperature is too high.
Turn the thermostat down and wait
5 minutes. Take the temperature
again and enter it.
Enter the new Celsius temperature: 103.2 Enter
The temperature is too high.
Turn the thermostat down and wait
5 minutes. Take the temperature
again and enter it.
Enter the new Celsius temperature: 102.1 Enter
The temperature is acceptable.
Check it again in 15 minutes.
```

Program Output (with input shown in bold)

```
Enter the substance's Celsius temperature: 102.1 Enter
The temperature is acceptable.
Check it again in 15 minutes.
```

Infinite Loops

In all but rare cases, loops must contain within themselves a way to terminate. This means that something inside the loop must eventually make the test condition false. The loop in Program 4-1 stops when the expression `keep_going == 'y'` is false. If a loop does not have a way of stopping, it is called an infinite loop. An *infinite loop* continues to repeat until the program is interrupted. Infinite loops usually occur when the programmer forgets to write code inside the loop that makes the test condition false. In most circumstances, you should avoid writing infinite loops.

Program 4-3 demonstrates an infinite loop. This is a modified version of the commission calculating program shown in Program 4-1. In this version, we have removed the code that modifies the `keep_going` variable in the body of the loop. Each time the expression `keep_going == 'y'` is tested in line 6, `keep_going` will reference the string `'y'`. As a consequence, the loop has no way of stopping. (The only way to stop this program is to press Ctrl+C on the keyboard to interrupt it.)

Program 4-3 (infinite.py)

```
 1  # This program demonstrates an infinite loop.
 2  # Create a variable to control the loop.
 3  keep_going = 'y'
 4
 5  # Warning! Infinite loop!
 6  while keep_going == 'y':
 7      # Get a salesperson's sales and commission rate.
 8      sales = float(input('Enter the amount of sales: '))
 9      comm_rate = float(input('Enter the commission rate: '))
10
11      # Calculate the commission.
12      commission = sales * comm_rate
13
14      # Display the commission.
15      print('The commission is $',
16            format(commission, ',.2f'), sep='')
```

Checkpoint

4.4 What is a loop iteration?

4.5 Does the `while` loop test its condition before or after it performs an iteration?

4.6 How many times will `'Hello World'` be printed in the following program?

```
count = 10
while count < 1:
      print('Hello World')
```

4.7 What is an infinite loop?

4.3 The `for` Loop: A Count-Controlled Loop

CONCEPT: A count-controlled loop iterates a specific number of times. In Python, you use the `for` statement to write a count-controlled loop.

VideoNote
The `for` Loop

As mentioned at the beginning of this chapter, a count-controlled loop iterates a specific number of times. Count-controlled loops are commonly used in programs. For example, suppose a business is open six days per week, and you are going to write a program that calculates the total sales for a week. You will need a loop that iterates exactly six times. Each time the loop iterates, it will prompt the user to enter the sales for one day.

You use the `for` statement to write a count-controlled loop. In Python, the `for` statement is designed to work with a sequence of data items. When the statement executes, it iterates once for each item in the sequence. Here is the general format:

```
for variable in [value1, value2, etc.]:
    statement
    statement
    etc.
```

We will refer to the first line as the *`for` clause*. In the `for` clause, *`variable`* is the name of a variable. Inside the brackets a sequence of values appears, with a comma separating each value. (In Python, a comma-separated sequence of data items that are enclosed in a set of brackets is called a *list*. In Chapter 7, you will learn more about lists.) Beginning at the next line is a block of statements that is executed each time the loop iterates.

The `for` statement executes in the following manner: The *`variable`* is assigned the first value in the list, then the statements that appear in the block are executed. Then, *`variable`* is assigned the next value in the list, and the statements in the block are executed again. This continues until *`variable`* has been assigned the last value in the list. Program 4-4 shows a simple example that uses a `for` loop to display the numbers 1 through 5.

Program 4-4 **(simple_loop1.py)**

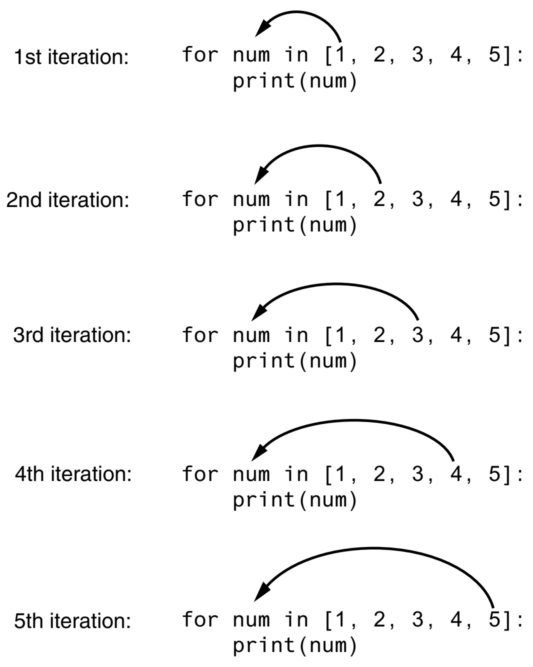
```
1  # This program demonstrates a simple for loop
2  # that uses a list of numbers.
3
4  print('I will display the numbers 1 through 5.')
5  for num in [1, 2, 3, 4, 5]:
6      print(num)
```

Program Output

```
I will display the numbers 1 through 5.
1
2
3
4
5
```

The first time the for loop iterates, the num variable is assigned the value 1 and then the statement in line 6 executes (displaying the value 1). The next time the loop iterates, num is assigned the value 2, and the statement in line 6 executes (displaying the value 2). This process continues, as shown in Figure 4-4, until num has been assigned the last value in the list. Because the list contains five values, the loop will iterate five times.

Figure 4-4 The for loop



```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

Python programmers commonly refer to the variable that is used in the `for` clause as the *target variable* because it is the target of an assignment at the beginning of each loop iteration.

The values that appear in the list do not have to be a consecutively ordered series of numbers. For example, Program 4-5 uses a `for` loop to display a list of odd numbers. There are five numbers in the list, so the loop iterates five times.

Program 4-5 (simple_loop2.py)

```
# This program also demonstrates a simple for
# loop that uses a list of numbers.

print('I will display the odd numbers 1 through 9.')
for num in [1, 3, 5, 7, 9]:
    print(num)
```

Program Output

```
I will display the odd numbers 1 through 9.
1
3
5
7
9
```

Program 4-6 shows another example. In this program, the `for` loop iterates over a list of strings. Notice the list (in line 4) contains the three strings 'Winken', 'Blinken', and 'Nod'. As a result, the loop iterates three times.

Program 4-6 (simple_loop3.py)

```
1  # This program also demonstrates a simple for
2  # loop that uses a list of strings.
3
4  for name in ['Winken', 'Blinken', 'Nod']:
5      print(name)
```

Program Output

```
Winken
Blinken
Nod
```

Using the `range` Function with the `for` Loop

Python provides a built-in function named `range` that simplifies the process of writing a count-controlled `for` loop. The range function creates a type of object known as an iterable. An *iterable* is an object that is similar to a list. It contains a sequence of values that can be iterated over with something like a loop. Here is an example of a for loop that uses the range function:

```
for num in range(5):
    print(num)
```

Notice instead of using a list of values, we call to the range function passing 5 as an argument. In this statement, the range function will generate an iterable sequence of integers in the range of 0 up to (but not including) 5. This code works the same as the following:

```
for num in [0, 1, 2, 3, 4]:
    print(num)
```

As you can see, the list contains five numbers, so the loop will iterate five times. Program 4-7 uses the range function with a for loop to display “Hello world” five times.

Program 4-7 (simple_loop4.py)

```
# This program demonstrates how the range
# function can be used with a for loop.

# Print a message five times.
for x in range(5):
    print('Hello world')
```

Program Output

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

If you pass one argument to the range function, as demonstrated in Program 4-7, that argument is used as the ending limit of the sequence of numbers. If you pass two arguments to the range function, the first argument is used as the starting value of the sequence, and the second argument is used as the ending limit. Here is an example:

```
for num in range(1, 5):
    print(num)
```

This code will display the following:

```
1
2
3
4
```

By default, the range function produces a sequence of numbers that increase by 1 for each successive number in the list. If you pass a third argument to the range function, that argument is used as *step value*. Instead of increasing by 1, each successive number in the sequence will increase by the step value. Here is an example:

```
for num in range(1, 10, 2):
    print(num)
```

In this `for` statement, three arguments are passed to the `range` function:

- The first argument, 1, is the starting value for the sequence.
- The second argument, 10, is the ending limit of the list. This means that the last number in the sequence will be 9.
- The third argument, 2, is the step value. This means that 2 will be added to each successive number in the sequence.

This code will display the following:

```
1
3
5
7
9
```

Using the Target Variable Inside the Loop

In a `for` loop, the purpose of the target variable is to reference each item in a sequence of items as the loop iterates. In many situations it is helpful to use the target variable in a calculation or other task within the body of the loop. For example, suppose you need to write a program that displays the numbers 1 through 10 and their respective squares, in a table similar to the following:

Number	Square
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

This can be accomplished by writing a `for` loop that iterates over the values 1 through 10. During the first iteration, the target variable will be assigned the value 1, during the second iteration it will be assigned the value 2, and so forth. Because the target variable will reference the values 1 through 10 during the loop's execution, you can use it in the calculation inside the loop. Program 4-8 shows how this is done.

Program 4-8 (squares.py)

```
 1  # This program uses a loop to display a
 2  # table showing the numbers 1 through 10
 3  # and their squares.
 4
 5  # Print the table headings.
 6  print('Number\tSquare')
 7  print('--------------')
 8
 9  # Print the numbers 1 through 10
10  # and their squares.
11  for number in range(1, 11):
12      square = number**2
13      print(number, '\t', square)
```

Program Output

```
Number  Square
--------------
1        1
2        4
3        9
4        16
5        25
6        36
7        49
8        64
9        81
10       100
```

First, take a closer look at line 6, which displays the table headings:

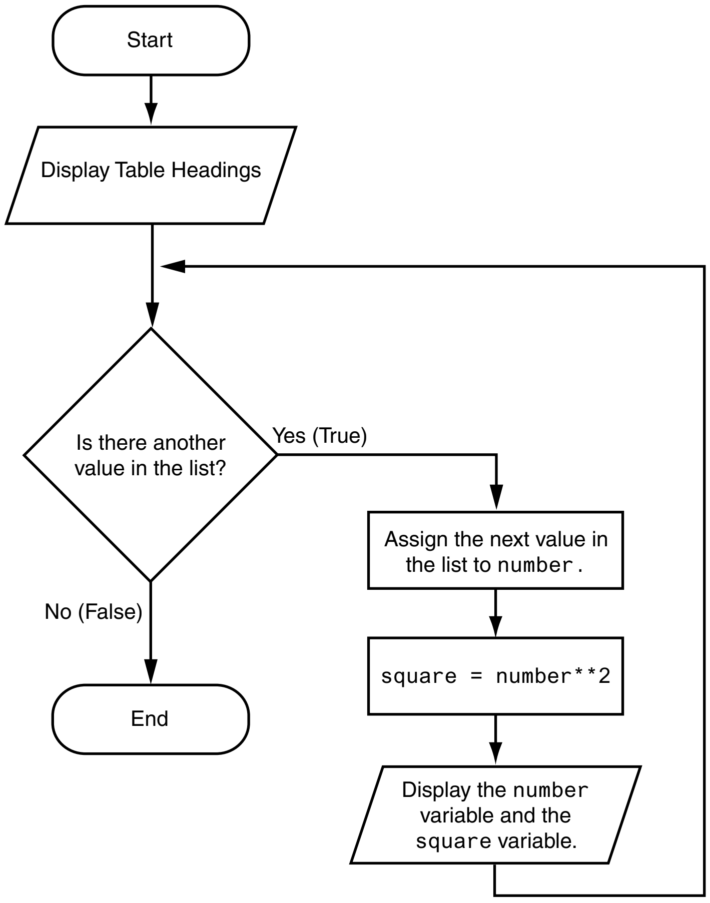
```
print('Number\tSquare')
```

Notice inside the string literal, the `\t` escape sequence between the words Number and Square. Recall from Chapter 2 that the `\t` escape sequence is like pressing the Tab key; it causes the output cursor to move over to the next tab position. This causes the space that you see between the words Number and Square in the sample output.

The `for` loop that begins in line 11 uses the `range` function to produce a sequence containing the numbers 1 through 10. During the first iteration, `number` will reference 1, during the second iteration `number` will reference 2, and so forth, up to 10. Inside the loop, the statement in line 12 raises `number` to the power of 2 (recall from Chapter 2 that `**` is the exponent operator) and assigns the result to the `square` variable. The statement in line 13 prints the value referenced by `number`, tabs over, then prints the value referenced by `square`. (Tabbing over with the `\t` escape sequence causes the numbers to be aligned in two columns in the output.)

Figure 4-5 shows how we might draw a flowchart for this program.

Figure 4-5 Flowchart for Program 4-8


Start
Display Table Headings
Is there another value in the list?
Yes (True)
Assign the next value in the list to number.
square = number**2
Display the number variable and the square variable.
No (False)
End

In the Spotlight:

Designing a Count-Controlled Loop with the for Statement

Your friend Amanda just inherited a European sports car from her uncle. Amanda lives in the United States, and she is afraid she will get a speeding ticket because the car's speedometer indicates kilometers per hour (KPH). She has asked you to write a program that displays a table of speeds in KPH with their values converted to miles per hour (MPH). The formula for converting KPH to MPH is:

$$MPH = KPH * 0.6214$$

In the formula, *MPH* is the speed in miles per hour, and *KPH* is the speed in kilometers per hour.

The table that your program displays should show speeds from 60 KPH through 130 KPH, in increments of 10, along with their values converted to MPH. The table should look something like this:

KPH	MPH
60	37.3
70	43.5
80	49.7
etc. . . .	
130	80.8

After thinking about this table of values, you decide that you will write a `for` loop. The list of values that the loop will iterate over will be the kilometer-per-hour speeds. In the loop, you will call the `range` function like this:

```
range(60, 131, 10)
```

The first value in the sequence will be 60. Notice the third argument specifies 10 as the step value. This means the numbers in the list will be 60, 70, 80, and so forth. The second argument specifies 131 as the sequence's ending limit, so the last number in the sequence will be 130.

Inside the loop, you will use the target variable to calculate a speed in miles per hour. Program 4-9 shows the program.

Program 4-9 (speed_converter.py)

```
# This program converts the speeds 60 kph
# through 130 kph (in 10 kph increments)
# to mph.

START_SPEED = 60             # Starting speed
END_SPEED = 131              # Ending speed
INCREMENT = 10               # Speed increment
CONVERSION_FACTOR = 0.6214   # Conversion factor

# Print the table headings.
print('KPH\tMPH')
print('--------------')

# Print the speeds.
for kph in range(START_SPEED, END_SPEED, INCREMENT)
    mph = kph * CONVERSION_FACTOR
    print(kph, '\t', format(mph, '.1f'))
```

(program continues)

Program 4-9 *(continued)*

Program Output

```
KPH        MPH
----------------
60         37.3
70         43.5
80         49.7
90         55.9
100        62.1
110        68.4
120        74.6
130        80.8
```

Letting the User Control the Loop Iterations

In many cases, the programmer knows the exact number of iterations that a loop must perform. For example, recall Program 4-8, which displays a table showing the numbers 1 through 10 and their squares. When the code was written, the programmer knew that the loop had to iterate over the values 1 through 10.

Sometimes, the programmer needs to let the user control the number of times that a loop iterates. For example, what if you want Program 4-8 to be a bit more versatile by allowing the user to specify the maximum value displayed by the loop? Program 4-10 shows how you can accomplish this.

Program 4-10 **(user_squares1.py)**

```
1   # This program uses a loop to display a
2   # table of numbers and their squares.
3
4   # Get the ending limit.
5   print('This program displays a list of numbers')
6   print('(starting at 1) and their squares.')
7   end = int(input('How high should I go? '))
8
9   # Print the table headings.
10  print()
11  print('Number\tSquare')
12  print('--------------')
13
14  # Print the numbers and their squares.
15  for number in range(1, end + 1):
16      square = number**2
17      print(number, '\t', square)
```

Program Output (with input shown in bold)

```
This program displays a list of numbers
(starting at 1) and their squares.
How high should I go? 5 Enter

Number   Square
-----------------
1        1
2        4
3        9
4        16
5        25
```

This program asks the user to enter a value that can be used as the ending limit for the list. This value is assigned to the `end` variable in line 7. Then, the expression `end + 1` is used in line 15 as the second argument for the `range` function. (We have to add one to `end` because otherwise the sequence would go up to, but not include, the value entered by the user.)

Program 4-11 shows an example that allows the user to specify both the starting value and the ending limit of the sequence.

Program 4-11 (user_squares2.py)

```python
# This program uses a loop to display a
# table of numbers and their squares.

# Get the starting value.
print('This program displays a list of numbers')
print('and their squares.')
start = int(input('Enter the starting number: '))

# Get the ending limit.
end = int(input('How high should I go? '))

# Print the table headings.
print()
print('Number\tSquare')
print('--------------')

# Print the numbers and their squares.
for number in range(start, end + 1):
    square = number**2
    print(number, '\t', square)
```

(program continues)

Program 4-11 *(continued)*

Program Output (with input shown in bold)

```
This program displays a list of numbers and their squares.
Enter the starting number: 5 Enter
How high should I go? 10 Enter

Number  Square
--------------
5        25
6        36
7        49
8        64
9        81
10       100
```

Generating an Iterable Sequence that Ranges from Highest to Lowest

In the examples you have seen so far, the `range` function was used to generate a sequence with numbers that go from lowest to highest. Alternatively, you can use the `range` function to generate sequences of numbers that go from highest to lowest. Here is an example:

```
range(10, 0, -1)
```

In this function call, the starting value is 10, the sequence's ending limit is 0, and the step value is −1. This expression will produce the following sequence:

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Here is an example of a `for` loop that prints the numbers 5 down to 1:

```
for num in range(5, 0, -1):
    print(num)
```

Checkpoint

4.8 Rewrite the following code so it calls the `range` function instead of using the list `[0, 1, 2, 3, 4, 5]`:

```
for x in [0, 1, 2, 3, 4, 5]:
    print('I love to program!')
```

4.9 What will the following code display?

```
for number in range(6):
    print(number)
```

4.10 What will the following code display?

```
for number in range(2, 6):
    print(number)
```

4.11 What will the following code display?

```
for number in range(0, 501, 100):
    print(number)
```

4.12 What will the following code display?

```
for number in range(10, 5, -1):
    print(number)
```

4.4 Calculating a Running Total

CONCEPT: A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator.

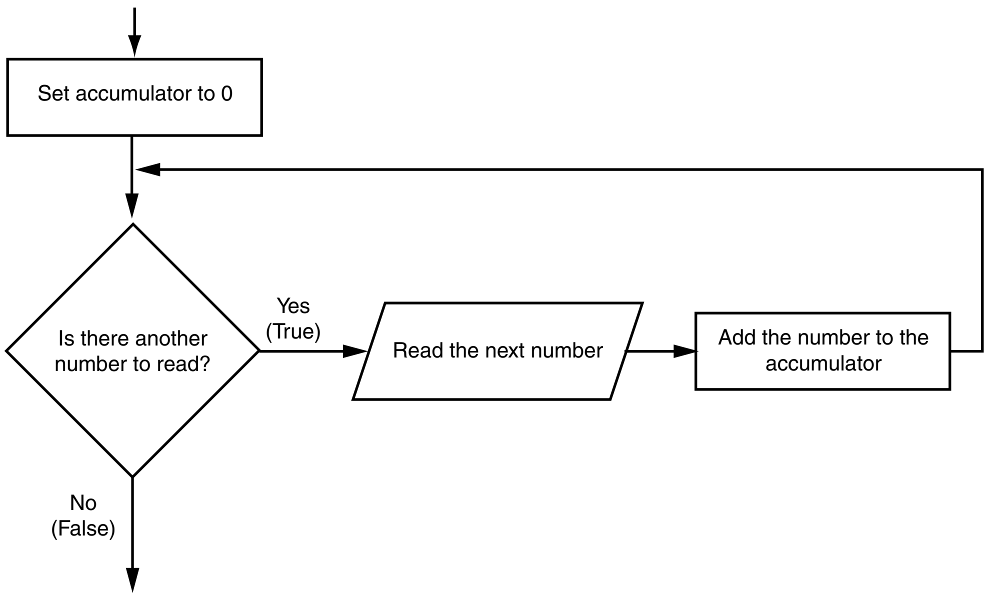
Many programming tasks require you to calculate the total of a series of numbers. For example, suppose you are writing a program that calculates a business's total sales for a week. The program would read the sales for each day as input and calculate the total of those numbers.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

The variable that is used to accumulate the total of the numbers is called an *accumulator*. It is often said that the loop keeps a *running total* because it accumulates the total as it reads each number in the series. Figure 4-6 shows the general logic of a loop that calculates a running total.

Figure 4-6 Logic for calculating a running total


Set accumulator to 0
Is there another number to read?
Yes (True)
Read the next number
Add the number to the accumulator
No (False)

When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop. Notice the first step in the flowchart is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

Let's look at a program that calculates a running total. Program 4-12 allows the user to enter five numbers, and displays the total of the numbers entered.

Program 4-12 (sum_numbers.py)

```
 1  # This program calculates the sum of a series
 2  # of numbers entered by the user.
 3
 4  MAX = 5 # The maximum number
 5
 6  # Initialize an accumulator variable.
 7  total = 0.0
 8
 9  # Explain what we are doing.
10  print('This program calculates the sum of')
11  print(MAX, 'numbers you will enter.')
12
13  # Get the numbers and accumulate them.
14  for counter in range(MAX):
15      number = int(input('Enter a number: '))
16      total = total + number
17
18  # Display the total of the numbers.
19  print('The total is', total)
```

Program Output (with input shown in bold)

```
This program calculates the sum of
5 numbers you will enter.
Enter a number: 1 Enter
Enter a number: 2 Enter
Enter a number: 3 Enter
Enter a number: 4 Enter
Enter a number: 5 Enter
The total is 15.0
```

The `total` variable, created by the assignment statement in line 7, is the accumulator. Notice it is initialized with the value 0.0. The `for` loop, in lines 14 through 16, does the work of getting the numbers from the user and calculating their total. Line 15 prompts the user to enter a number then assigns the input to the `number` variable. Then, the following statement in line 16 adds `number` to `total`:

```
total = total + number
```

After this statement executes, the value referenced by the `number` variable will be added to the value in the `total` variable. It's important that you understand how this statement works. First, the interpreter gets the value of the expression on the right side of the `=` operator, which is `total + number`. Then, that value is assigned by the `=` operator to the `total` variable. The effect of the statement is that the value of the `number` variable is added to the `total` variable. When the loop finishes, the `total` variable will hold the sum of all the numbers that were added to it. This value is displayed in line 19.

The Augmented Assignment Operators

Quite often, programs have assignment statements in which the variable that is on the left side of the = operator also appears on the right side of the = operator. Here is an example:

```
x = x + 1
```

On the right side of the assignment operator, 1 is added to `x`. The result is then assigned to `x`, replacing the value that `x` previously referenced. Effectively, this statement adds 1 to `x`. You saw another example of this type of statement in Program 4-13:

```
total = total + number
```

This statement assigns the value of `total + number` to `total`. As mentioned before, the effect of this statement is that `number` is added to the value of `total`. Here is one more example:

```
balance = balance - withdrawal
```

This statement assigns the value of the expression `balance - withdrawal` to `balance`. The effect of this statement is that `withdrawal` is subtracted from `balance`.

Table 4-1 shows other examples of statements written this way.

Table 4-1 Various assignment statements (assume x = 6 in each statement)

Statement	What It Does	Value of x after the Statement
`x = x + 4`	Add 4 to `x`	10
`x = x − 3`	Subtracts 3 from `x`	3
`x = x * 10`	Multiplies `x` by 10	60
`x = x / 2`	Divides `x` by 2	3
`x = x % 4`	Assigns the remainder of `x / 4` to `x`	2

These types of operations are common in programming. For convenience, Python offers a special set of operators designed specifically for these jobs. Table 4-2 shows the *augmented assignment operators*.

Table 4-2 Augmented assignment operators

Operator	Example Usage	Equivalent To
`+=`	`x += 5`	`x = x + 5`
`−=`	`y −= 2`	`y = y − 2`
`*=`	`z *= 10`	`z = z * 10`
`/=`	`a /= b`	`a = a / b`
`%=`	`c %= 3`	`c = c % 3`

As you can see, the augmented assignment operators do not require the programmer to type the variable name twice. The following statement:

```
total = total + number
```

could be rewritten as

```
total += number
```

Similarly, the statement

```
balance = balance - withdrawal
```

could be rewritten as

```
balance -= withdrawal
```

Checkpoint

4.13 What is an accumulator?

4.14 Should an accumulator be initialized to any specific value? Why or why not?

4.15 What will the following code display?

```
total = 0
for count in range(1, 6):
    total = total + count
print(total)
```

4.16 What will the following code display?

```
number 1 = 10
number 2 = 5
number 1 = number 1 + number 2
print(number1)
print(number2)
```

4.17 Rewrite the following statements using augmented assignment operators:

a) `quantity = quantity + 1`
b) `days_left = days_left - 5`
c) `price = price * 10`
d) `price = price / 2`

4.5 Sentinels

CONCEPT: A sentinel is a special value that marks the end of a sequence of values.

Consider the following scenario: You are designing a program that will use a loop to process a long sequence of values. At the time you are designing the program, you do not know the number of values that will be in the sequence. In fact, the number of values in the sequence could be different each time the program is executed. What is the best way to design such a loop? Here are some techniques that you have seen already in this chapter, along with the disadvantages of using them when processing a long list of values:

- Simply ask the user, at the end of each loop iteration, if there is another value to process. If the sequence of values is long, however, asking this question at the end of each loop iteration might make the program cumbersome for the user.
- Ask the user at the beginning of the program how many items are in the sequence. This might also inconvenience the user, however. If the sequence is very long, and the user does not know the number of items it contains, it will require the user to count them.

When processing a long sequence of values with a loop, perhaps a better technique is to use a sentinel. A *sentinel* is a special value that marks the end of a sequence of items. When a program reads the sentinel value, it knows it has reached the end of the sequence, so the loop terminates.

For example, suppose a doctor wants a program to calculate the average weight of all her patients. The program might work like this: A loop prompts the user to enter either a patient's weight, or 0 if there are no more weights. When the program reads 0 as a weight, it interprets this as a signal that there are no more weights. The loop ends and the program displays the average weight.

A sentinel value must be distinctive enough that it will not be mistaken as a regular value in the sequence. In the example cited above, the doctor (or her medical assistant) enters 0 to signal the end of the sequence of weights. Because no patient's weight will be 0, this is a good value to use as a sentinel.

In the Spotlight:
Using a Sentinel

The county tax office calculates the annual taxes on property using the following formula:

$$\textit{property tax} = \textit{property value} \times 0.0065$$

Every day, a clerk in the tax office gets a list of properties and has to calculate the tax for each property on the list. You have been asked to design a program that the clerk can use to perform these calculations.

In your interview with the tax clerk, you learn that each property is assigned a lot number, and all lot numbers are 1 or greater. You decide to write a loop that uses the number 0 as a sentinel value. During each loop iteration, the program will ask the clerk to enter either a property's lot number, or 0 to end. The code for the program is shown in Program 4-13.

Program 4-13 (property_tax.py)

```
# This program displays property taxes.

TAX_FACTOR = 0.0065 # Represents the tax factor.

```

(program continues)

Program 4-13 *(continued)*

```
# Get the first lot number.
print('Enter the property lot number')
print('or enter 0 to end.')
lot = int(input('Lot number: '))

# Continue processing as long as the user
# does not enter lot number 0.
while lot ! = 0:
    # Get the property value.
    value = float(input('Enter the property value: '))

    # Calculate the property's tax.
    tax = value * TAX_FACTOR

    # Display the tax.
    print('Property tax: $', format(tax, ',.2f'), sep='')

    # Get the next lot number.
    print('Enter the next lot number or')
    print('enter 0 to end.')
    lot = int(input('Lot number: '))
```

Program Output (with input shown in bold)

```
Enter the property lot number
or enter 0 to end.
Lot number: 100 Enter
Enter the property value: 100000.00 Enter
Property tax: $650.00.
Enter the next lot number or
enter 0 to end.
Lot number: 200 Enter
Enter the property value: 5000.00 Enter
Property tax: $32.50.
Enter the next lot number or
enter 0 to end.
Lot number: 0 Enter
```

Checkpoint

4.18 What is a sentinel?

4.19 Why should you take care to choose a distinctive value as a sentinel?

4.6 Input Validation Loops

CONCEPT: Input validation is the process of inspecting data that has been input to a program, to make sure it is valid before it is used in a computation. Input validation is commonly done with a loop that iterates as long as an input variable references bad data.

One of the most famous sayings among computer programmers is “garbage in, garbage out.” This saying, sometimes abbreviated as *GIGO*, refers to the fact that computers cannot tell the difference between good data and bad data. If a user provides bad data as input to a program, the program will process that bad data and, as a result, will produce bad data as output. For example, look at the payroll program in Program 4-14 and notice what happens in the sample run when the user gives bad data as input.

Program 4-14 (gross_pay.py)

```
# This program displays gross pay.
# Get the number of hours worked.
hours = int(input('Enter the hours worked this week: '))

# Get the hourly pay rate.
pay_rate = float(input('Enter the hourly pay rate: '))

# Calculate the gross pay.
gross_pay = hours * pay_rate

# Display the gross pay.
print('Gross pay: $', format(gross_pay, ',.2f'))
```

Program Output (with input shown in bold)

```
Enter the hours worked this week: 400 Enter
Enter the hourly pay rate: 20 Enter
The gross pay is $8,000.00
```

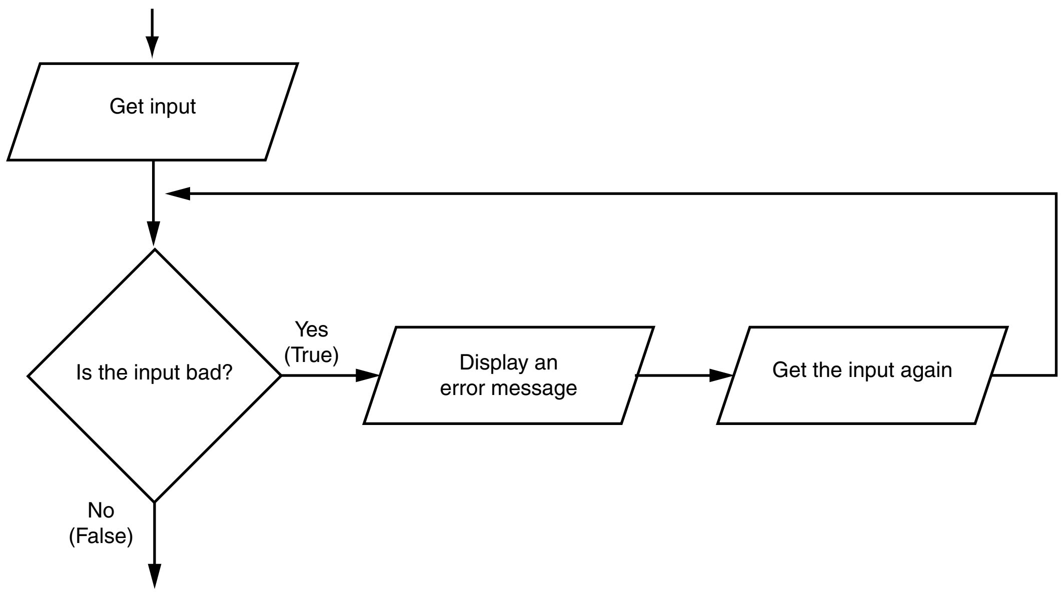
Did you spot the bad data that was provided as input? The person receiving the paycheck will be pleasantly surprised, because in the sample run the payroll clerk entered 400 as the number of hours worked. The clerk probably meant to enter 40, because there are not 400 hours in a week. The computer, however, is unaware of this fact, and the program processed the bad data just as if it were good data. Can you think of other types of input that can be given to this program that will result in bad output? One example is a negative number entered for the hours worked; another is an invalid hourly pay rate.

Sometimes stories are reported in the news about computer errors that mistakenly cause people to be charged thousands of dollars for small purchases, or to receive large tax refunds to which they were not entitled. These "computer errors" are rarely caused by the computer, however; they are more commonly caused by bad data that was read into a program as input.

The integrity of a program's output is only as good as the integrity of its input. For this reason, you should design your programs in such a way that bad input is never accepted. When input is given to a program, it should be inspected before it is processed. If the input is invalid, the program should discard it and prompt the user to enter the correct data. This process is known as *input validation*.

Figure 4-7 shows a common technique for validating an item of input. In this technique, the input is read, then a loop is executed. If the input data is bad, the loop executes its block of statements. The loop displays an error message so the user will know that the input was invalid, and then it reads the new input. The loop repeats as long as the input is bad.

Figure 4-7 Logic containing an input validation loop



Notice the flowchart in Figure 4-7 reads input in two places: first just before the loop, and then inside the loop. The first input operation—just before the loop—is called a *priming read*, and its purpose is to get the first input value that will be tested by the validation loop. If that value is invalid, the loop will perform subsequent input operations.

Let's consider an example. Suppose you are designing a program that reads a test score and you want to make sure the user does not enter a value less than 0. The following code shows how you can use an input validation loop to reject any input value that is less than 0.

```
# Get a test score.
score = int(input('Enter a test score: '))
```

```
# Make sure it is not less than 0.
while score < 0:
    print('ERROR: The score cannot be negative.')
    score = int(input('Enter the correct score: '))
```

This code first prompts the user to enter a test score (this is the priming read), then the `while` loop executes. Recall that the `while` loop is a pretest loop, which means it tests the expression `score < 0` before performing an iteration. If the user entered a valid test score, this expression will be false, and the loop will not iterate. If the test score is invalid, however, the expression will be true, and the loop's block of statements will execute. The loop displays an error message and prompts the user to enter the correct test score. The loop will continue to iterate until the user enters a valid test score.

NOTE: An input validation loop is sometimes called an *error trap* or an *error handler.*

This code rejects only negative test scores. What if you also want to reject any test scores that are greater than 100? You can modify the input validation loop so it uses a compound Boolean expression, as shown next.

```
# Get a test score.
score = int(input('Enter a test score: '))
# Make sure it is not less than 0 or greater than 100.
while score < 0 or score > 100:
    print('ERROR: The score cannot be negative')
    print('or greater than 100.')
    score = int(input('Enter the correct score: '))
```

The loop in this code determines whether `score` is less than 0 or greater than 100. If either is true, an error message is displayed and the user is prompted to enter a correct score.

In the Spotlight:
Writing an Input Validation Loop

Samantha owns an import business, and she calculates the retail prices of her products with the following formula:

$$\textit{retail price} = \textit{wholesale cost} \times 2.5$$

She currently uses the program shown in Program 4-15 to calculate retail prices.

Program 4-15 (`retail_no_validation.py`)

```python
# This program calculates retail prices.

MARK_UP = 2.5 # The markup percentage
another = 'y' # Variable to control the loop.

# Process one or more items.
while another == 'y' or another == 'Y':
    # Get the item's wholesale cost.
    wholesale = float(input("Enter the item's " +
                            "wholesale cost: "))

    # Calculate the retail price.
    retail = wholesale * MARK_UP

    # Display the retail price.
    print('Retail price: $', format(retail, ',.2f'), sep='')


    # Do this again?
    another = input('Do you have another item? ' +
                    '(Enter y for yes): ')
```

Program Output (with input shown in bold)

```
Enter the item's wholesale cost: 10.00 Enter
Retail price: $25.00.
Do you have another item? (Enter y for yes): y Enter
Enter the item's wholesale cost: 15.00 Enter
Retail price: $37.50.
Do you have another item? (Enter y for yes): y Enter
Enter the item's wholesale cost: 12.50 Enter
Retail price: $31.25.
Do you have another item? (Enter y for yes): n Enter
```

Samantha has encountered a problem when using the program, however. Some of the items that she sells have a wholesale cost of 50 cents, which she enters into the program as 0.50. Because the 0 key is next to the key for the negative sign, she sometimes accidentally enters a negative number. She has asked you to modify the program so it will not allow a negative number to be entered for the wholesale cost.

You decide to add an input validation loop to the `show_retail` function that rejects any negative numbers that are entered into the `wholesale` variable. Program 4-16 shows the revised program, with the new input validation code shown in lines 13 through 16.

Program 4-16 (retail_with_validation.py)

```python
# This program calculates retail prices.

MARK_UP = 2.5 # The markup percentage
another = 'y' # Variable to control the loop.

# Process one or more items.
while another == 'y' or another == 'Y':
    # Get the item's wholesale cost.
    wholesale = float(input("Enter the item's " +
                            "wholesale cost: "))

    # Validate the wholesale cost.
    while wholesale < 0:
        print('ERROR: the cost cannot be negative.')
        wholesale = float(input('Enter the correct' +
                                'wholesale cost: '))

    # Calculate the retail price.
    retail = wholesale * MARK_UP

    # Display the retail price.
    print('Retail price: $', format(retail, ',.2f'), sep='')


    # Do this again?
    another = input('Do you have another item? ' +
                    '(Enter y for yes): ')
```

Program Output (with input shown in bold)

```
Enter the item's wholesale cost: -.50 Enter
ERROR: the cost cannot be negative.
Enter the correct wholesale cost: 0.50 Enter
Retail price: $1.25.
Do you have another item? (Enter y for yes): n Enter
```

Checkpoint

4.20 What does the phrase "garbage in, garbage out" mean?

4.21 Give a general description of the input validation process.

4.22 Describe the steps that are generally taken when an input validation loop is used to validate data.

4.23 What is a priming read? What is its purpose?

4.24 If the input that is read by the priming read is valid, how many times will the input validation loop iterate?

4.7 Nested Loops

CONCEPT: A loop that is inside another loop is called a nested loop.

A nested loop is a loop that is inside another loop. A clock is a good example of something that works like a nested loop. The second hand, minute hand, and hour hand all spin around the face of the clock. The hour hand, however, only makes 1 revolution for every 12 of the minute hand's revolutions. And it takes 60 revolutions of the second hand for the minute hand to make 1 revolution. This means that for every complete revolution of the hour hand, the second hand has revolved 720 times. Here is a loop that partially simulates a digital clock. It displays the seconds from 0 to 59:

```
for seconds in range(60):
    print(seconds)
```

We can add a `minutes` variable and nest the loop above inside another loop that cycles through 60 minutes:

```
for minutes in range(60):
    for seconds in range(60):
        print(minutes, ':', seconds)
```

To make the simulated clock complete, another variable and loop can be added to count the hours:

```
for hours in range(24):
    for minutes in range(60):
        for seconds in range(60):
            print(hours, ':', minutes, ':', seconds)
```

This code's output would be:

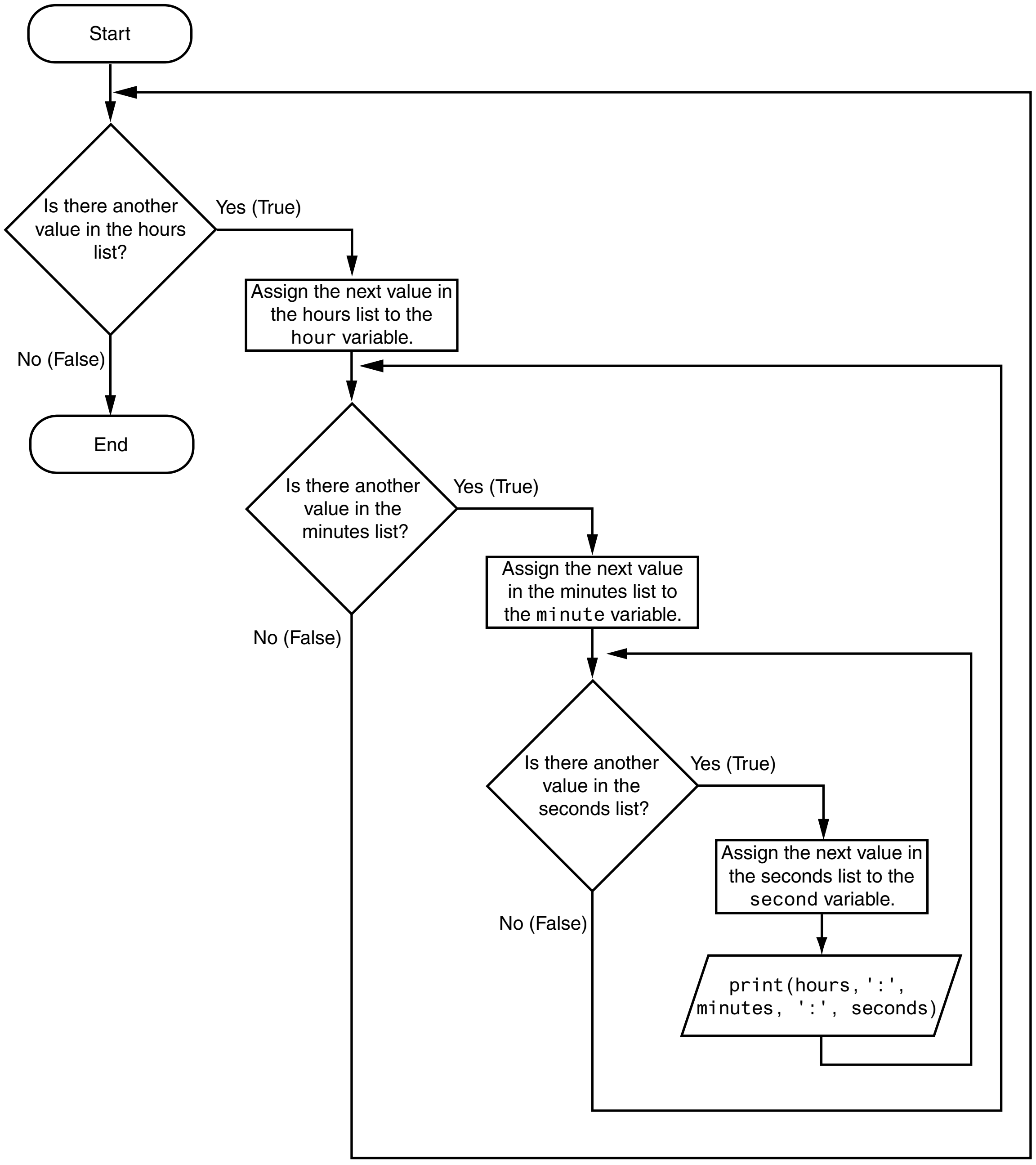
```
0 : 0 : 0
0 : 0 : 1
0 : 0 : 2
```

(The program will count through each second of 24 hours.)

```
23 : 59 : 59
```

The innermost loop will iterate 60 times for each iteration of the middle loop. The middle loop will iterate 60 times for each iteration of the outermost loop. When the outermost loop has iterated 24 times, the middle loop will have iterated 1,440 times and the innermost loop will have iterated 86,400 times! Figure 4-8 shows a flowchart for the complete clock simulation program previously shown.

Figure 4-8 Flowchart for a clock simulator


Start
Is there another value in the hours list?
Yes (True)
Assign the next value in the hours list to the hour variable.
No (False)
End
Is there another value in the minutes list?
Yes (True)
Assign the next value in the minutes list to the minute variable.
No (False)
Is there another value in the seconds list?
Yes (True)
Assign the next value in the seconds list to the second variable.
No (False)
print(hours, ':', minutes, ':', seconds)

The simulated clock example brings up a few points about nested loops:

- An inner loop goes through all of its iterations for every single iteration of an outer loop.
- Inner loops complete their iterations faster than outer loops.

- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops.

Program 4-17 shows another example. It is a program that a teacher might use to get the average of each student's test scores. The statement in line 5 asks the user for the number of students, and the statement in line 8 asks the user for the number of test scores per student. The for loop that begins in line 11 iterates once for each student. The nested inner loop, in lines 17 through 21, iterates once for each test score.

Program 4-17 (test_score_averages.py)

```
 1  # This program averages test scores. It asks the user for the
 2  # number of students and the number of test scores per student.
 3
 4  # Get the number of students.
 5  num_students = int(input('How many students do you have? '))
 6
 7  # Get the number of test scores per student.
 8  num_test_scores = int(input('How many test scores per student? '))
 9
10  # Determine each student's average test score.
11  for student in range(num_students):
12      # Initialize an accumulator for test scores.
13      total = 0.0
14      # Get a student's test scores.
15      print('Student number', student + 1)
16      print('-----------------')
17      for test_num in range(num_test_scores):
18          print('Test number', test_num + 1, end='')
19          score = float(input(': '))
20          # Add the score to the accumulator.
21          total += score
22
23      # Calculate the average test score for this student.
24      average = total / num_test_scores
25
26      # Display the average.
27      print('The average for student number', student + 1,
28            'is:', average)
29      print()
```

Program Output (with input shown in bold)

```
How many students do you have? 3 Enter
How many test scores per student? 3 Enter
```

```
Student number 1
-----------------
Test number 1: 100 Enter
Test number 2: 95 Enter
Test number 3: 90 Enter
The average for student number 1 is: 95.0

Student number 2
-----------------
Test number 1: 80 Enter
Test number 2: 81 Enter
Test number 3: 82 Enter
The average for student number 2 is: 81.0

Student number 3
-----------------
Test number 1: 75 Enter
Test number 2: 85 Enter
Test number 3: 80 Enter
The average for student number 3 is: 80.0
```

In the Spotlight:

Using Nested Loops to Print Patterns

One interesting way to learn about nested loops is to use them to display patterns on the screen. Let's look at a simple example. Suppose we want to print asterisks on the screen in the following rectangular pattern:

```
******
******
******
******
******
******
******
******
```

If you think of this pattern as having rows and columns, you can see that it has eight rows, and each row has six columns. The following code can be used to display one row of asterisks:

```
for col in range(6):
    print('*', end='')
```

If we run this code in a program or in interactive mode, it produces the following output:

```
******
```

To complete the entire pattern, we need to execute this loop eight times. We can place the loop inside another loop that iterates eight times, as shown here:

```
1  for row in range(8):
2      for col in range(6):
3          print('*', end='')
4      print()
```

The outer loop iterates eight times. Each time it iterates, the inner loop iterates 6 times. (Notice in line 4, after each row has been printed, we call the `print()` function. We have to do that to advance the screen cursor to the next line at the end of each row. Without that statement, all the asterisks will be printed in one long row on the screen.)

We could easily write a program that prompts the user for the number of rows and columns, as shown in Program 4-18.

Program 4-18 (rectangluar_pattern.py)

```
# This program displays a rectangular pattern
# of asterisks.
rows = int(input('How many rows? '))
cols = int(input('How many columns? '))

for r in range(rows):
    for c in range(cols):
        print('*', end='')
    print()
```

Program Output (with input shown in bold)

```
How many rows? 5 Enter
How many columns? 10 Enter
**********
**********
**********
**********
**********
```

Let's look at another example. Suppose you want to print asterisks in a pattern that looks like the following triangle:

```
*
**
***
****
*****
******
*******
********
```

Once again, think of the pattern as being arranged in rows and columns. The pattern has a total of eight rows. In the first row, there is one column. In the second row, there are two columns. In the third row, there are three columns. This continues to the eighth row, which has eight columns. Program 4-19 shows the code that produces this pattern.

Program 4-19 (triangle_pattern.py)

```
1  # This program displays a triangle pattern.
2  BASE_SIZE = 8
3
4  for r in range(BASE_SIZE):
5      for c in range(r + 1):
6          print('*', end='')
7      print()
```

Program Output

```
*
**
***
****
*****
******
*******
********
```

First, let's look at the outer loop. In line 4, the expression `range(BASE_SIZE)` produces an iterable containing the following sequence of integers:

```
0, 1, 2, 3, 4, 5, 6, 7
```

As a result, the variable `r` is assigned the values 0 through 7 as the outer loop iterates. The inner loop's `range` expression, in line 5, is `range(r + 1)`. The inner loop executes as follows:

- During the outer loop's first iteration, the variable `r` is assigned 0. The expression `range(r + 1)` causes the inner loop to iterate one time, printing one asterisk.
- During the outer loop's second iteration, the variable `r` is assigned 1. The expression `range(r + 1)` causes the inner loop to iterate two times, printing two asterisks.
- During the outer loop's third iteration, the variable `r` is assigned 2. The expression `range(r + 1)` causes the inner loop to iterate three times, printing three asterisks, and so forth.

Let's look at another example. Suppose you want to display the following stair-step pattern:

```
#
 #
  #
   #
    #
     #
```

The pattern has six rows. In general, we can describe each row as having some number of spaces followed by a # character. Here's a row-by-row description:

First row:	0 spaces followed by a # character.
Second row:	1 space followed by a # character.
Third row:	2 spaces followed by a # character.
Fourth row:	3 spaces followed by a # character.
Fifth row:	4 spaces followed by a # character.
Sixth row:	5 spaces followed by a # character.

To display this pattern, we can write code containing a pair of nested loops that work in the following manner:

- The outer loop will iterate six times. Each iteration will perform the following:
  - The inner loop will display the correct number of spaces, side by side.
  - Then, a # character will be displayed.

Program 4-20 shows the Python code.

Program 4-20 **(stair_step_pattern.py)**

```
1  # This program displays a stair-step pattern.
2  NUM_STEPS = 6
3
4  for r in range(NUM_STEPS):
5      for c in range(r):
6          print(' ', end='')
7      print('#')
```

Program Output

```
#
 #
  #
   #
    #
     #
```

In line 1, the expression `range(NUM_STEPS)` produces an iterable containing the following sequence of integers:

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

As a result, the outer loop iterates 6 times. As the outer loop iterates, variable `r` is assigned the values 0 through 5. The inner loop executes as follows:

- During the outer loop's first iteration, the variable `r` is assigned 0. A loop that is written as `for c in range(0):` iterates zero times, so the inner loop does not execute at this time.
- During the outer loop's second iteration, the variable `r` is assigned 1. A loop that is written as `for c in range(1):` iterates one time, so the inner loop iterates once, printing one space.
- During the outer loop's third iteration, the variable `r` is assigned 2. A loop that is written as `for c in range(2):` will iterate two times, so the inner loop iterates twice, printing two spaces, and so forth.

4.8 Turtle Graphics: Using Loops to Draw Designs

CONCEPT: You can use loops to draw graphics that range in complexity from simple shapes to elaborate designs.

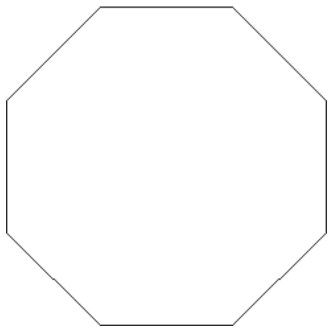
You can use loops with the turtle to draw both simple shapes and elaborate designs. For example, the following `for` loop iterates four times to draw a square that is 100 pixels wide:

```
for x in range(4):
    turtle.forward(100)
    turtle.right(90)
```

The following code shows another example. This `for` loop iterates eight times to draw the octagon shown in Figure 4-9:

```
for x in range(8):
    turtle.forward(100)
    turtle.right(45)
```

Figure 4-9 Octagon



Program 4-21 shows an example of using a loop to draw concentric circles. The program's output is shown in Figure 4-10.

Program 4-21 **(concentric_circles.py)**

```
# Concentric circles
import turtle

# Named constants
NUM_CIRCLES = 20
STARTING_RADIUS = 20
```

(program continues)