Chapter 8

Performing Repetitive Tasks

In This Chapter

- Performing a task a specific number of times
- Performing a task until completion
- Placing one task loop within another

All the examples in the book so far have performed a series of steps just one time and then stopped. However, the real world doesn't work this way. Many of the tasks that humans perform are repetitious. For example, the doctor might state that you need to exercise more and tell you to do 100 push-ups each day. If you just do one push-up, you won't get much benefit from the exercise and you definitely won't be following the doctor's orders. Of course, because you know precisely how many push-ups to do, you can perform the task a specific number of times. Python allows the same sort of repetition using the `for` statement.

Unfortunately, you don't always know how many times to perform a task. For example, consider needing to check a stack of coins for one of extreme rarity. Taking just the first coin from the top, examining it, and deciding that it either is or isn't the rare coin doesn't complete the task. Instead, you must examine each coin in turn, looking for the rare coin. Your stack may contain more than one. Only after you have looked at every coin in the stack can you say that the task is complete. However, because you don't know how many coins are in the stack, you don't know how many times to perform the task at the outset. You only know the task is done when the stack is gone. Python performs this kind of repetition using the `while` statement.

Most programming languages call any sort of repeating sequence of events a *loop.* The idea is to picture the repetition as a circle, with the code going round and round executing tasks until the loop ends. Loops are an essential part of application elements such as menus. In fact, writing most modern applications without using loops would be impossible.

In some cases, you must create loops within loops. For example, to create a multiplication table, you use a loop within a loop. The inner loop calculates the column values and the outer loop moves between rows. You see such an example later in the chapter, so don't worry too much about understanding precisely how such things work right now.

Processing Data Using the for Statement

The first looping code block that most developers encounter is the `for` statement. It's hard to imagine creating a conventional programming language that lacks such a statement. In this case, the loop executes a fixed number of times, and you know the number of times it will execute before the loop even begins. Because everything about a `for` loop is known at the outset, `for` loops tend to be the easiest kind of loop to use. However, in order to use one, you need to know how many times to execute the loop. The following sections describe the `for` loop in greater detail.

Understanding the for statement

A `for` loop begins with a `for` statement. The `for` statement describes how to perform the loop. The Python `for` loop works through a sequence of some type. It doesn't matter whether the sequence is a series of letters in a string or items within a collection. You can even specify a range of values to use by specifying the `range()` function. Here's a simple `for` statement.

```
for Letter in "Howdy!":
```

The statement begins with the keyword `for`. The next item is a variable that holds a single element of a sequence. In this case, the variable name is `Letter`. The `in` keyword tells Python that the sequence comes next. In this case, the sequence is the string `"Howdy"`. The `for` statement always ends with a colon, just as the decision-making statements described in Chapter 7 do.

Indented under the `for` statement are the tasks you want performed within the `for` loop. Python considers every following indented statement part of the code block that composes the `for` loop. Again, the `for` loop works just like the decision-making statements in Chapter 7.

Creating a basic for loop

The best way to see how a `for` loop actually works is to create one. In this case, the example uses a string for the sequence. The `for` loop processes each of the characters in the string in turn until it runs out of characters. This example also appears with the downloadable source code as `SimpleFor.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
LetterNum = 1

for Letter in "Howdy!":
   print("Letter ", LetterNum, " is ", Letter)
   LetterNum+=1
```

   The example begins by creating a variable, `LetterNum`, to track the number of letters that have been processed. Every time the loop completes, `LetterNum` is updated by 1.

   The `for` statement works through the sequence of letters in the string `"Howdy!"`. It places each letter, in turn, in `Letter`. The code that follows displays the current `LetterNum` value and its associated character found in `Letter`.

3. **Choose Run⇨Run Module.**

   A Python Shell window opens. The application displays the letter sequence along with the letter number, as shown in Figure 8-1.

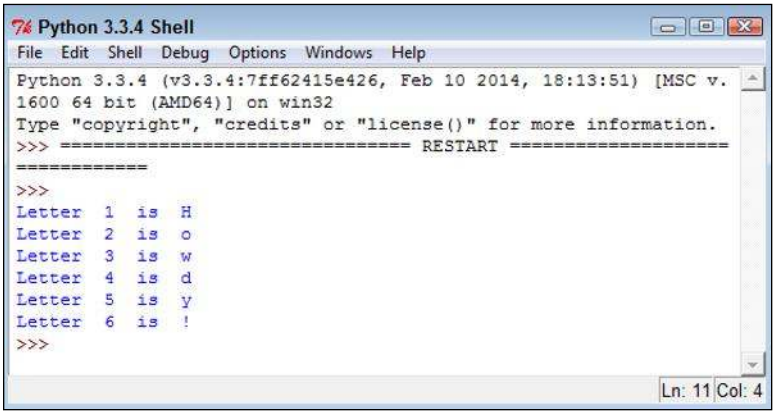

Figure 8-1: Use the `for` loop to process the characters in a string one at a time.

Controlling execution with the break statement

Life is often about exceptions to the rule. For example, you might want an assembly line to produce a number of clocks. However, at some point, the assembly line runs out of a needed part. If the part isn't available, the assembly line must stop in the middle of the processing cycle. The count hasn't completed, but the line must be stopped anyway until the missing part is restocked.

Interruptions also occur in computers. You might be streaming data from an online source when a network glitch occurs and breaks the connection; the stream temporarily runs dry, so the application runs out of things to do even though the set number of tasks isn't completed.

The `break` clause makes breaking out of a loop possible. However, you don't simply place the `break` clause in your code — you surround it with an `if` statement that defines the condition for issuing a `break`. The statement might say something like this: If the stream runs dry, then break out of the loop.

In this example, you see what happens when the count reaches a certain level when processing a string. The example is a little contrived in the interest of keeping things simple, but it reflects what could happen in the real world when a data element is too long to process (possibly indicating an error condition). This example also appears with the downloadable source code as `ForBreak.py`.

1. **Open a Python File window.**

   You see an editor where you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
Value = input("Type less than 6 characters: ")
LetterNum = 1

for Letter in Value:
   print("Letter ", LetterNum, " is ", Letter)
   LetterNum+=1
   if LetterNum > 6:
      print("The string is too long!")
      break
```

This example builds on the one found in the previous section. However, it lets the user provide a variable-length string. When the string is longer than six characters, the application stops processing it.

The `if` statement contains the conditional code. When `LetterNum` is greater than 6, it means that the string is too long. Notice the second level of indentation used for the `if` statement. In this case, the user sees an error message stating that the string is too long, and then the code executes a `break` to end the loop.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open with a prompt asking for input.

4. **Type** Hello **and press Enter.**

   The application lists each character in the string, as shown in Figure 8-2.

5. **Perform Steps 3 and 4 again, but type** I am too long. **instead of Hello.**

   The application displays the expected error message and stops processing the string at character 6, as shown in Figure 8-3.

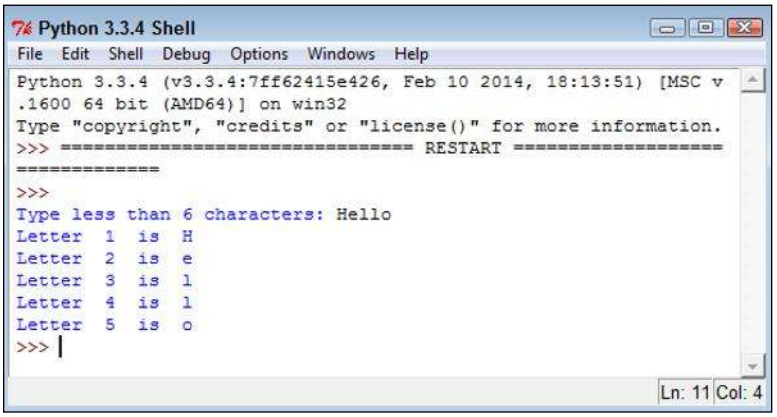

Figure 8-2: A short string is successfully processed by the application.

This example adds *length checking* to your repertoire of application data error checks. Chapter 7 shows how to perform range checks, which ensure that a value meets specific limits. The length check is necessary to ensure that data, especially strings, aren't going to overrun the size of data fields. In addition, a small input size makes it harder for intruders to perform certain types of hacks on your system, which makes your system more secure.

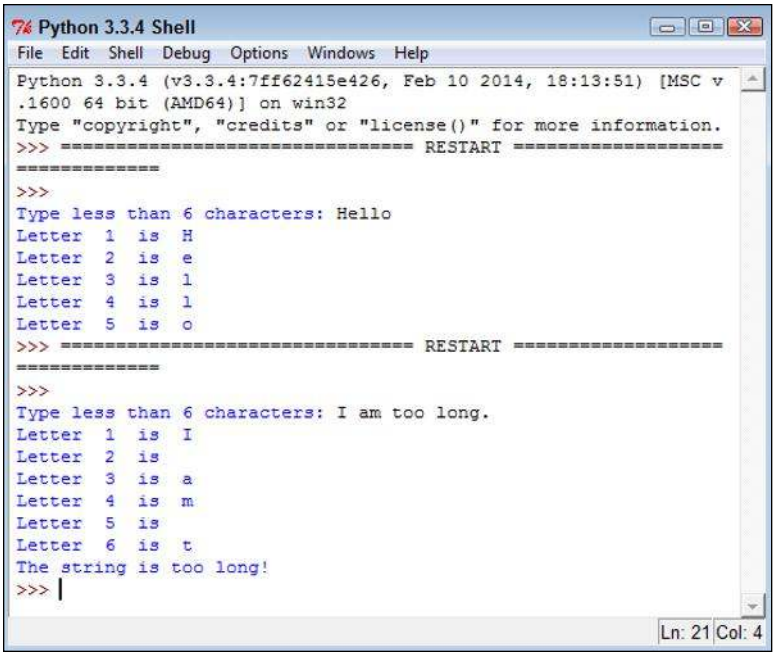

Figure 8-3: Long strings are truncated to ensure that they remain a certain size.

Controlling execution with the continue statement

Sometimes you want to check every element in a sequence, but don't want to process certain elements. For example, you might decide that you want to process all the information for every car in a database except brown cars. Perhaps you simply don't need the information about that particular color of car. The `break` clause simply ends the loop, so you can't use it in this situation. Otherwise, you won't see the remaining elements in the sequence.

The `break` clause alternative that many developers use is the `continue` clause. As with the `break` clause, the `continue` clause appears as part of an `if` statement. However, processing continues with the next element in the sequence rather than ending completely.

The following steps help you see how the `continue` clause differs from the `break` clause. In this case, the code refuses to process the letter *w*, but will process every other letter in the alphabet. This example also appears with the downloadable source code as `ForContinue.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
LetterNum = 1

for Letter in "Howdy!":
   if Letter == "w":
      continue
      print("Encountered w, not processed.")
   print("Letter ", LetterNum, " is ", Letter)
   LetterNum+=1
```

   This example is based on the one found in the "Creating a basic for loop" section, earlier in this chapter. However, this example adds an `if` statement with the `continue` clause in the `if` code block. Notice the `print()` function that is part of the `if` code block. You never see this string printed because the current loop iteration ends immediately.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application displays the letter sequence along with the letter number, as shown in Figure 8-4. However, notice the effect of the `continue` clause — the letter *w* isn't processed.

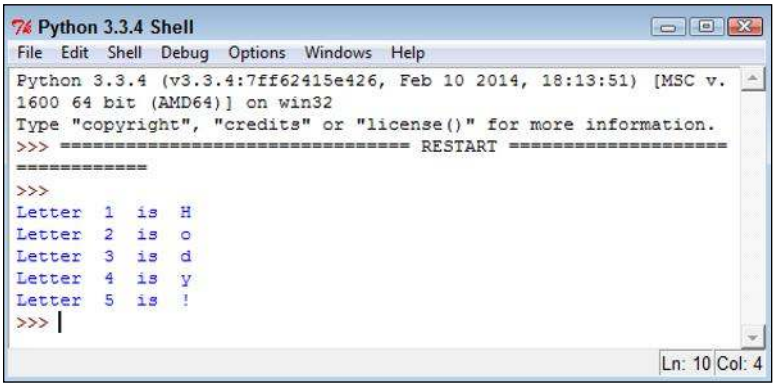

Figure 8-4: Use the `continue` clause to avoid processing specific elements.

Controlling execution with the pass clause

The Python language includes something not commonly found in other languages: a second sort of `continue` clause. The `pass` clause works almost the same way as the `continue` clause does, except that it allows completion of the code in the `if` code block in which it appears. The following steps use an example that is precisely the same as the one found in the previous section, "Controlling execution with the continue statement," except that it uses a `pass` clause instead. This example also appears with the downloadable source code as `ForPass.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
LetterNum = 1

for Letter in "Howdy!":
   if Letter == "w":
      pass
      print("Encountered w, not processed.")
   print("Letter ", LetterNum, " is ", Letter)
   LetterNum+=1
```

3. **Choose Run➪Run Module.**

   You see a Python Shell window open. The application displays the letter sequence along with the letter number, as shown in Figure 8-5. However, notice the effect of the `pass` clause — the letter *w* isn't processed. In addition, the example displays the string that wasn't displayed for the `continue` clause example.

The `continue` clause makes it possible to silently bypass specific elements in a sequence and to avoid executing any additional code for that element. Use the `pass` clause when you need to perform some sort of post processing on the element, such as logging the element in an error log, displaying a message to the user, or handling the problem element in some other way. The `continue` and `pass` clauses both do the same thing, but they're used in distinctly different situations.

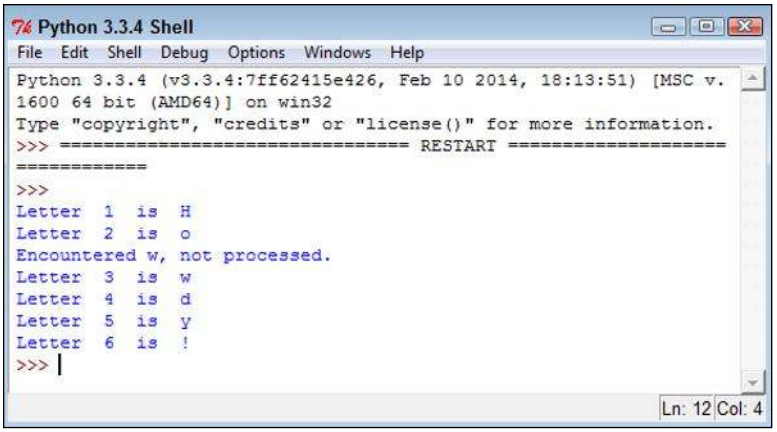

Figure 8-5: Using the `pass` clause allows for post processing of an unwanted input.

Controlling execution with the else statement

Python has another loop clause that you won't find with other languages: `else`. The `else` clause makes executing code possible even if you have no elements to process in a sequence. For example, you might need to convey to the user that there simply isn't anything to do. In fact, that's what the following example does. This example also appears with the downloadable source code as `ForElse.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
Value = input("Type less than 6 characters: ")
LetterNum = 1

for Letter in Value:
   print("Letter ", LetterNum, " is ", Letter)
   LetterNum+=1
else:
   print("The string is blank.")
```

This example is based on the one found in the "Creating a basic for loop" section, earlier in the chapter. However, when a user presses Enter without typing something, the `else` clause is executed.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open and a prompt asking for input.

4. **Type** Hello **and press Enter.**

   The application lists each character in the string, as shown in Figure 8-2.

5. **Repeat Steps 3 and 4. However, simply press Enter instead of entering any sort of text.**

   You see the alternative message shown in Figure 8-6 that tells you the string is blank.

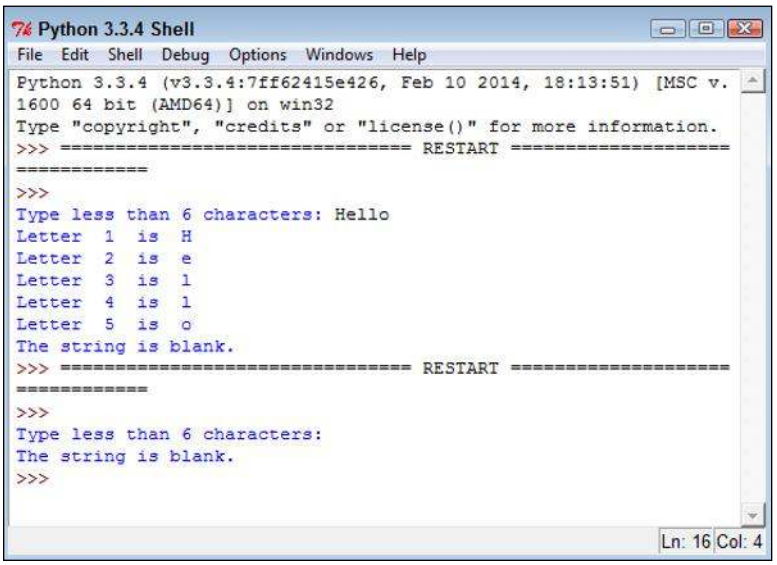

Figure 8-6: The `else` clause makes it possible to perform tasks based on an empty sequence.

It's easy to misuse the `else` clause because an empty sequence doesn't always signify a simple lack of input. An empty sequence could also signal an application error or other conditions that need to be handled differently from a simple omission of data. Make sure you understand how the application works with data to ensure that the `else` clause doesn't end up hiding potential error conditions, rather than making them visible so that they can be fixed.

Processing Data Using the while Statement

You use the `while` statement for situations when you're not sure how much data the application will have to process. Instead of instructing Python to process a static number of items, you use the `while` statement to tell Python to continue processing items until it runs out of items. This kind of loop is useful when you need to perform tasks such as downloading files of unknown size or streaming data from a source such as a radio station. Any situation in which you can't define at the outset how much data the application will process is a good candidate for the `while` statement that is described more fully in the sections that follow.

Understanding the while statement

The `while` statement works with a condition rather than a sequence. The condition states that the `while` statement should perform a task until the condition is no longer true. For example, imagine a deli with a number of customers standing in front of the counter. The salesperson continues to service customers until no more customers are left in line. The line could (and probably will) grow as the other customers are handled, so it's impossible to know at the outset how many customers will be served. All the salesperson knows is that continuing to serve customers until no more are left is important. Here is how a `while` statement might look:

```
while Sum < 5:
```

The statement begins with the `while` keyword. It then adds a condition. In this case, a variable, `Sum`, must be less than `5` for the loop to continue. Nothing specifies the current value of `Sum`, nor does the code define how the value of `Sum` will change. The only thing that is known when Python executes the statement is that `Sum` must be less than `5` for the loop to continue performing tasks. The statement ends with a colon and the tasks are indented below the statement.

Because the `while` statement doesn't perform a series of tasks a set number of times, creating an *endless loop* is possible, meaning that the loop never ends. For example, say that `Sum` is set to `0` when the loop begins, and the ending condition is that `Sum` must be less than `5`. If the value of `Sum` never increases, the loop will continue executing forever (or at least until the computer is shut down). Endless loops can cause all sorts of bizarre problems on systems, such as slowdowns and even computer freezes, so it's best to avoid

them. You must always provide a method for the loop to end when using a `while` loop (contrasted with the `for` loop, in which the end of the sequence determines the end of the loop). So, when working with the `while` statement, you must perform three tasks:

1. Create the environment for the condition (such as setting `Sum` to `0`).
2. State the condition within the `while` statement (such as `Sum < 5`).
3. Update the condition as needed to ensure that the loop eventually ends (such as adding `Sum+=1` to the `while` code block).

As with the `for` statement, you can modify the default behavior of the `while` statement. In fact, you have access to the same four clauses to modify the `while` statement behavior:

- `break`: Ends the current loop.
- `continue`: Immediately ends processing of the current element.
- `pass`: Ends processing of the current element after completing the statements in the `if` block.
- `else`: Provides an alternative processing technique when conditions aren't met for the loop.

Using the while statement in an application

You can use the `while` statement in many ways, but this first example is straightforward. It simply displays a count based on the starting and ending condition of a variable named `Sum`. The following steps help you create and test the example code. This example also appears with the downloadable source code as `SimpleWhile.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
Sum = 0

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

The example code demonstrates the three tasks you must perform when working with a `while` loop in a straightforward manner. It begins by setting `Sum` to `0`, which is the first step of setting the condition environment. The condition itself appears as part of the `while` statement. The end of the `while` code block accomplishes the third step. Of course, the code displays the current value of `Sum` before it updates the value of `Sum`.

A `while` statement provides flexibility that you don't get with a `for` statement. This example shows a relatively straightforward way to update `Sum`. However, you can use any update method required to meet the goals of the application. Nothing says that you have to update `Sum` in a specific manner. In addition, the condition can be as complex as you want it to be. For example, you can track the current value of three or four variables if so desired. Of course, the more complex you make the condition, the more likely it is that you'll create an endless loop, so you have a practical limit as to how complex you should make the `while` loop condition.

3. **Choose Run⇨Run Module.**

   Python executes the `while` loop and displays the numeric sequence shown in Figure 8-7.

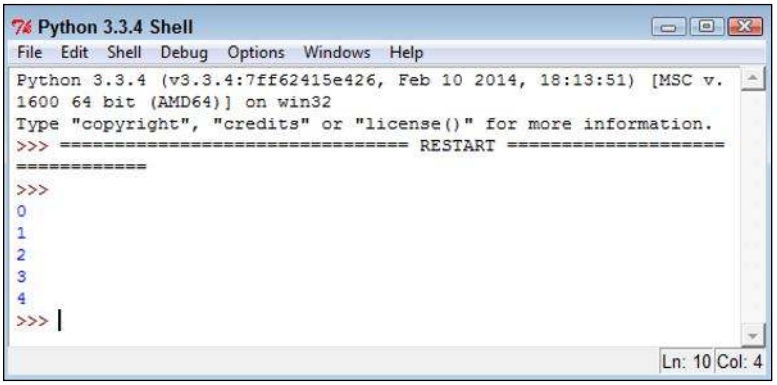

Figure 8-7: The simple `while` loop displays a sequence of numbers.

Nesting Loop Statements

In some cases, you can use either a `for` loop or a `while` loop to achieve the same effect. The manners work differently, but the effect is the same. In this example, you create a multiplication table generator by nesting a `while` loop within a `for` loop. Because you want the output to look nice, you use a little formatting as well (Chapter 11 provides you with detailed instruction in this regard). This example also appears with the downloadable source code as `ForElse.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
X = 1
Y = 1

print ('{:>4}'.format(' '), end= ' ')

for X in range(1, 11):
   print('{:>4}'.format(X), end=' ')

print()

for X in range(1,11):
   print('{:>4}'.format(X), end=' ')
   while Y <= 10:
      print('{:>4}'.format(X * Y), end=' ')
      Y+=1
   print()
   Y=1
```

This example begins by creating two variables, `X` and `Y`, to hold the row and column value of the table. `X` is the row variable and `Y` is the column variable.

To make the table readable, this example must create a heading at the top and another along the side. When users see a 1 at the top and a 1 at the side, and follow these values to where they intersect in the table, they can see the value of the two numbers when multiplied.

The first `print()` statement adds a space (because nothing appears in the corner of the table; see Figure 8-8 to more easily follow this discussion). All the formatting statement says is to create a space 4 characters wide and place a space within it. The `{:>4}` part of the code determines the size of the column. The `format(' ')` function determines what appears in that space. The end attribute of the `print()` statement changes the ending character from a carriage return to a simple space.

The first `for` loop displays the numbers 1 through 10 at the top of the table. The `range()` function creates the sequence of numbers for you. When using the `range()` function, you specify the starting value, which is `1` in this case, and one more than the ending value, which is `11` in this case.

At this point, the cursor is sitting at the end of the heading row. To move it to the next line, the code issues a `print()` call with no other information.

Even though the next bit of code looks quite complex, you can figure it out if you look at it a line at a time. The multiplication table shows the values from 1 * 1 to 10 * 10, so you need ten rows and ten columns to display the information. The `for` statement tells Python to create ten rows.

Look again at Figure 8-8 to note the row heading. The first `print()` call displays the row heading value. Of course, you have to format this information, and the code uses a space of four characters that end with a space, rather than a carriage return, in order to continue printing information in that row.

The `while` loop comes next. This loop prints the columns in an individual row. The column values are the multiplied values of `X * Y`. Again, the output is formatted to take up four spaces. The `while` loop ends when `Y` is updated to the next value using `Y+=1`.

Now you're back into the `for` loop. The `print()` statement ends the current row. In addition, `Y` must be reset to `1` so that it's ready for the beginning of the next row, which begins with `1`.

3. **Choose Run⇨Run Module.**

   You see the multiplication table shown in Figure 8-8.

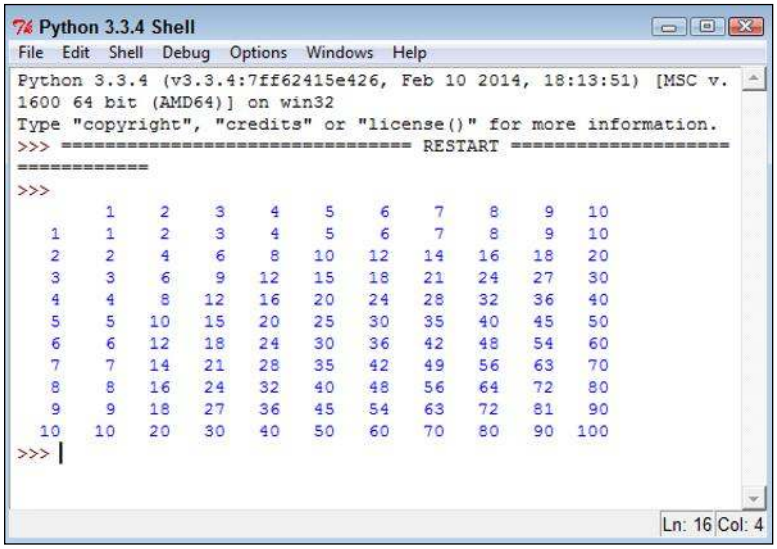

Figure 8-8: The multiplication table is pleasing to the eye thanks to its formatting.