PART III

Statements and Syntax

CHAPTER 10

Introducing Python Statements

Now that you're familiar with Python's core built-in object types, this chapter begins our exploration of its fundamental statement forms. As in the previous part, we'll begin here with a general introduction to statement syntax, and we'll follow up with more details about specific statements in the next few chapters.

In simple terms, *statements* are the things you write to tell Python what your programs should do. If programs "do things with stuff," statements are the way you specify what sort of things a program does. Python is a procedural, statement-based language; by combining statements, you specify a procedure that Python performs to satisfy a program's goals.

Python Program Structure Revisited

Another way to understand the role of statements is to revisit the concept hierarchy introduced in Chapter 4, which talked about built-in objects and the expressions used to manipulate them. This chapter climbs the hierarchy to the next level:

1. Programs are composed of modules.
2. Modules contain statements.
3. *Statements contain expressions*.
4. Expressions create and process objects.

At its core, Python syntax is composed of statements and expressions. Expressions process objects and are embedded in statements. Statements code the larger *logic* of a program's operation—they use and direct expressions to process the objects we studied in the preceding chapters. Moreover, statements are where objects spring into existence (e.g., in expressions within assignment statements), and some statements create entirely new kinds of objects (functions, classes, and so on). Statements always exist in modules, which themselves are managed with statements.

Python's Statements

Table 10-1 summarizes Python's statement set. This part of the book deals with entries in the table from the top through `break` and `continue`. You've informally been introduced to a few of the statements in Table 10-1 already; this part of the book will fill in details that were skipped earlier, introduce the rest of Python's procedural statement set, and cover the overall syntax model. Statements lower in Table 10-1 that have to do with larger program units—functions, classes, modules, and exceptions—lead to larger programming ideas, so they will each have a section of their own. More focused statements (like `del`, which deletes various components) are covered elsewhere in the book, or in Python's standard manuals.

Table 10-1. Python 3.0 statements

Statement	Role	Example
Assignment	Creating references	`a, *b = 'good', 'bad', 'ugly'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
`print` calls	Printing objects	`print('The Killer', joke)`
`if/elif/else`	Selecting actions	`if "python" in text:` `  print(text)`
`for/else`	Sequence iteration	`for x in mylist:` `  print(x)`
`while/else`	General loops	`while X > Y:` `  print('hello')`
`pass`	Empty placeholder	`while True:` `  pass`
`break`	Loop exit	`while True:` `  if exittest(): break`
`continue`	Loop continue	`while True:` `  if skiptest(): continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):` `  for i in n: yield i*2`

`global`	Namespaces	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

`nonlocal`	Namespaces (3.0+)	`def outer():`
		`  x = 'old'`
		`  def function():`
		`  nonlocal x; x = 'new'`

`import`	Module access	`import sys`
`from`	Attribute access	`from sys import stdin`

`class`	Building objects	`class Subclass(Superclass):`
		`  staticData = []`
		`  def method(self): pass`

Statement	Role	Example

try/except/finally	Catching exceptions	try:
		    action()
		except:
		    print('action error')

raise	Triggering exceptions	raise EndSearch(location)
assert	Debugging checks	assert X > Y, 'X too small'
with/as	Context managers (2.6+)	with open('data') as myfile: process(myfile)
del	Deleting references	del data[k] del data[i:j] del obj.attr del variable

Table 10-1 reflects the statement forms in Python 3.0—units of code that each have a specific syntax and purpose. Here are a few fine points about its content:

- Assignment statements come in a variety of syntax flavors, described in Chapter 11: basic, sequence, augmented, and more.
- `print` is technically neither a reserved word nor a statement in 3.0, but a built-in function call; because it will nearly always be run as an expression statement, though (that is, on a line by itself), it's generally thought of as a statement type. We'll study print operations in Chapter 11 the next chapter.
- `yield` is actually an expression instead of a statement too, as of 2.5; like `print`, it's typically used in a line by itself and so is included in this table, but scripts occasionally assign or otherwise use its result, as we'll see in Chapter 20. As an expression, `yield` is also a reserved word, unlike `print`.

Most of this table applies to Python 2.6, too, except where it doesn't—if you are using Python 2.6 or older, here are a few notes for your Python, too:

- In 2.6, `nonlocal` is not available; as we'll see in Chapter 17, there are alternative ways to achieve this statement's writeable state-retention effect.
- In 2.6, `print` is a statement instead of a built-in function call, with specific syntax covered in Chapter 11.
- In 2.6, the 3.0 `exec` code execution built-in function is a statement, with specific syntax; since it supports enclosing parentheses, though, you can generally use its 3.0 call form in 2.6 code.
- In 2.5, the `try`/`except` and `try`/`finally` statements were merged: the two were formerly separate statements, but we can now say both `except` and `finally` in the same `try` statement.
- In 2.5, `with`/`as` is an optional extension, and it is not available unless you explicitly turn it on by running the statement `from __future__ import with_statement` (see Chapter 33).

A Tale of Two ifs

Before we delve into the details of any of the concrete statements in Table 10-1, I want to begin our look at Python statement syntax by showing you what you are *not* going to type in Python code so you can compare and contrast it with other syntax models you might have seen in the past.

Consider the following `if` statement, coded in a C-like language:

```
if (x > y) {
    x = 1;
    y = 2;
}
```

This might be a statement in C, C++, Java, JavaScript, or Perl. Now, look at the equivalent statement in the Python language:

```
if x > y:
    x = 1
    y = 2
```

The first thing that may pop out at you is that the equivalent Python statement is less, well, cluttered—that is, there are fewer syntactic components. This is by design; as a scripting language, one of Python's goals is to make programmers' lives easier by requiring less typing.

More specifically, when you compare the two syntax models, you'll notice that Python adds one new thing to the mix, and that three items that are present in the C-like language are not present in Python code.

What Python Adds

The one new syntax component in Python is the colon character (`:`). All Python *compound statements* (i.e., statements that have statements nested inside them) follow the same general pattern of a header line terminated in a colon, followed by a nested block of code usually indented underneath the header line, like this:

```
Header line:
    Nested statement block
```

The colon is required, and omitting it is probably the most common coding mistake among new Python programmers—it's certainly one I've witnessed thousands of times in Python training classes. In fact, if you are new to Python, you'll almost certainly forget the colon character very soon. Most Python-friendly editors make this mistake easy to spot, and including it eventually becomes an unconscious habit (so much so that you may start typing colons in your C++ code, too, generating many entertaining error messages from your C++ compiler!).

What Python Removes

Although Python requires the extra colon character, there are three things programmers in C-like languages must include that you don't generally have to in Python.

Parentheses are optional

The first of these is the set of parentheses around the tests at the top of the statement:

```
if (x < y)
```

The parentheses here are required by the syntax of many C-like languages. In Python, though, they are not—we simply omit the parentheses, and the statement works the same way:

```
if x < y
```

Technically speaking, because every expression can be enclosed in parentheses, including them will not hurt in this Python code, and they are not treated as an error if present. *But don't do that*: you'll be wearing out your keyboard needlessly, and broadcasting to the world that you're an ex-C programmer still learning Python (I was once, too). The Python way is to simply omit the parentheses in these kinds of statements altogether.

End of line is end of statement

The second and more significant syntax component you won't find in Python code is the semicolon. You don't need to terminate statements with semicolons in Python the way you do in C-like languages:

```
x = 1;
```

In Python, the general rule is that the end of a line automatically terminates the statement that appears on that line. In other words, you can leave off the semicolons, and it works the same way:

```
x = 1
```

There are some ways to work around this rule, as you'll see in a moment. But, in general, you write one statement per line for the vast majority of Python code, and no semicolon is required.

Here, too, if you are pining for your C programming days (if such a state is possible...) you can continue to use semicolons at the end of each statement—the language lets you get away with them if they are present. *But don't do that either* (really!); again, doing so tells the world that you're still a C programmer who hasn't quite made the switch to Python coding. The Pythonic style is to leave off the semicolons altogether.

End of indentation is end of block

The third and final syntax component that Python removes, and the one that may seem the most unusual to soon-to-be-ex-C programmers (until they've used it for 10 minutes and realize it's actually a feature), is that you do not type anything explicit in your code to syntactically mark the beginning and end of a nested block of code. You don't need to include `begin`/`end`, `then`/`endif`, or braces around the nested block, as you do in C-like languages:

```
if (x > y) {
    x = 1;
    y = 2;
}
```

Instead, in Python, we consistently indent all the statements in a given single nested block the same distance to the right, and Python uses the statements' physical indentation to determine where the block starts and stops:

```
if x > y:
    x = 1
    y = 2
```

By *indentation*, I mean the blank whitespace all the way to the left of the two nested statements here. Python doesn't care how you indent (you may use either spaces or tabs), or how much you indent (you may use any number of spaces or tabs). In fact, the indentation of one nested block can be totally different from that of another. The syntax rule is only that for a given single nested block, all of its statements must be indented the same distance to the right. If this is not the case, you will get a syntax error, and your code will not run until you repair its indentation to be consistent.

Why Indentation Syntax?

The indentation rule may seem unusual at first glance to programmers accustomed to C-like languages, but it is a deliberate feature of Python, and it's one of the main ways that Python almost forces programmers to produce uniform, regular, and readable code. It essentially means that you must line up your code vertically, in columns, according to its logical structure. The net effect is to make your code more consistent and readable (unlike much of the code written in C-like languages).

To put that more strongly, aligning your code according to its logical structure is a major part of making it readable, and thus reusable and maintainable, by yourself and others. In fact, even if you never use Python after reading this book, you should get into the habit of aligning your code for readability in any block-structured language. Python forces the issue by making this a part of its syntax, but it's an important thing to do in any programming language, and it has a huge impact on the usefulness of your code.

Your experience may vary, but when I was still doing development on a full-time basis, I was mostly paid to work on large old C++ programs that had been worked on by many programmers over the years. Almost invariably, each programmer had his or her

own style for indenting code. For example, I'd often be asked to change a `while` loop coded in the C++ language that began like this:

```
while (x > 0) {
```

Before we even get into indentation, there are three or four ways that programmers can arrange these braces in a C-like language, and organizations often have political debates and write standards manuals to address the options (which seems more than a little off-topic for the problem to be solved by programming). Ignoring that, here's the scenario I often encountered in C++ code. The first person who worked on the code indented the loop four spaces:

```
while (x > 0) {
    --------;
    --------;
```

That person eventually moved on to management, only to be replaced by someone who liked to indent further to the right:

```
while (x > 0) {
    --------;
    --------;
           --------;
           --------;
```

That person later moved on to other opportunities, and someone else picked up the code who liked to indent less:

```
while (x > 0) {
    --------;
    --------;
           --------;
           --------;
--------;
--------;
}
```

And so on. Eventually, the block is terminated by a closing brace (}), which of course makes this "block-structured code" (he says, sarcastically). In any block-structured language, Python or otherwise, if nested blocks are not indented consistently, they become very difficult for the reader to interpret, change, or reuse, because the code no longer visually reflects its logical meaning. Readability matters, and indentation is a major component of readability.

Here is another example that may have burned you in the past if you've done much programming in a C-like language. Consider the following statement in C:

```
if (x)
     if (y)
          statement1;
else
     statement2;
```

Which `if` does the `else` here go with? Surprisingly, the `else` is paired with the nested `if` statement (`if (y)`), even though it looks visually as though it is associated with the outer `if (x)`. This is a classic pitfall in the C language, and it can lead to the reader completely misinterpreting the code and changing it incorrectly in ways that might not be uncovered until the Mars rover crashes into a giant rock!

This cannot happen in Python—because indentation is significant, the way the code looks is the way it will work. Consider an equivalent Python statement:

```
if x:
     if y:
          statement1
else:
     statement2
```

In this example, the `if` that the `else` lines up with vertically is the one it is associated with logically (the outer `if x`). In a sense, Python is a WYSIWYG language—what you see is what you get because the way code looks is the way it runs, regardless of who coded it.

If this still isn't enough to underscore the benefits of Python's syntax, here's another anecdote. Early in my career, I worked at a successful company that developed systems software in the C language, where consistent indentation is not required. Even so, when we checked our code into source control at the end of the day, this company ran an automated script that analyzed the indentation used in the code. If the script noticed that we'd indented our code inconsistently, we received an automated email about it the next morning—and so did our managers!

The point is that even when a language doesn't require it, good programmers know that consistent use of indentation has a huge impact on code readability and quality. The fact that Python promotes this to the level of syntax is seen by most as a feature of the language.

Also keep in mind that nearly every programmer-friendly text editor has built-in support for Python's syntax model. In the IDLE Python GUI, for example, lines of code are automatically indented when you are typing a nested block; pressing the Backspace key backs up one level of indentation, and you can customize how far to the right IDLE indents statements in a nested block. There is no universal standard on this: four spaces or one tab per level is common, but it's up to you to decide how and how much you wish to indent. Indent further to the right for further nested blocks, and less to close the prior block.

As a rule of thumb, you probably shouldn't mix tabs and spaces in the same block in Python, unless you do so consistently; use tabs or spaces in a given block, but not both (in fact, Python 3.0 now issues an error for inconsistent use of tabs and spaces, as we'll see in Chapter 12). But you probably shouldn't mix tabs or spaces in indentation in *any* structured language—such code can cause major readability issues if the next programmer has his or her editor set to display tabs differently than yours. C-like languages

might let coders get away with this, but they shouldn't: the result can be a mangled mess.

I can't stress enough that regardless of which language you code in, you should be indenting consistently for readability. In fact, if you weren't taught to do this earlier in your career, your teachers did you a disservice. Most programmers—especially those who must read others' code—consider it a major asset that Python elevates this to the level of syntax. Moreover, generating tabs instead of braces is no more difficult in practice for tools that must output Python code. In general, if you do what you should be doing in a C-like language anyhow, but get rid of the braces, your code will satisfy Python's syntax rules.

A Few Special Cases

As mentioned previously, in Python's syntax model:

- The end of a line terminates the statement on that line (without semicolons).
- Nested statements are blocked and associated by their physical indentation (without braces).

Those rules cover almost all Python code you'll write or see in practice. However, Python also provides some special-purpose rules that allow customization of both statements and nested statement blocks.

Statement rule special cases

Although statements normally appear one per line, it is possible to squeeze more than one statement onto a single line in Python by separating them with semicolons:

```
a = 1; b = 2; print(a + b)              # Three statements on one line
```

This is the only place in Python where semicolons are required: as *statement separators*. This only works, though, if the statements thus combined are not themselves compound statements. In other words, you can chain together only simple statements, like assignments, `print`s, and function calls. Compound statements must still appear on lines of their own (otherwise, you could squeeze an entire program onto one line, which probably would not make you very popular among your coworkers!).

The other special rule for statements is essentially the inverse: you can make a single statement span across multiple lines. To make this work, you simply have to enclose part of your statement in a bracketed pair—parentheses (`()`), square brackets (`[]`), or curly braces (`{}`). Any code enclosed in these constructs can cross multiple lines: your statement doesn't end until Python reaches the line containing the closing part of the pair. For instance, to continue a list literal:

```
mlist = [111,
         222,
         333]
```

Because the code is enclosed in a square brackets pair, Python simply drops down to the next line until it encounters the closing bracket. The curly braces surrounding dictionaries (as well as set literals and dictionary and set comprehensions in 3.0) allow them to span lines this way too, and parentheses handle tuples, function calls, and expressions. The indentation of the continuation lines does not matter, though common sense dictates that the lines should be aligned somehow for readability.

Parentheses are the catchall device—because any expression can be wrapped up in them, simply inserting a left parenthesis allows you to drop down to the next line and continue your statement:

```
X = (A + B +
     C + D)
```

This technique works with compound statements, too, by the way. Anywhere you need to code a large expression, simply wrap it in parentheses to continue it on the next line:

```
if (A == 1 and
    B == 2 and
    C == 3):
        print('spam' * 3)
```

An older rule also allows for continuation lines when the prior line ends in a backslash:

```
X = A + B + \                 # An error-prone alternative
      C + D
```

This alternative technique is dated, though, and is frowned on today because it's difficult to notice and maintain the backslashes, and it's fairly brittle—there can be no spaces after the backslash, and omitting it can have unexpected effects if the next line is mistaken to be a new statement. It's also another throwback to the C language, where it is commonly used in "#define" macros; again, when in Pythonland, do as Pythonistas do, not as C programmers do.

Block rule special case

As mentioned previously, statements in a nested block of code are normally associated by being indented the same amount to the right. As one special case here, the body of a compound statement can instead appear on the same line as the header in Python, after the colon:

```
if x > y: print(x)
```

This allows us to code single-line `if` statements, single-line loops, and so on. Here again, though, this will work only if the body of the compound statement itself does not contain any compound statements. That is, only simple statements—assignments, `print`s, function calls, and the like—are allowed after the colon. Larger statements must still appear on lines by themselves. Extra parts of compound statements (such as the `else` part of an `if`, which we'll meet later) must also be on separate lines of their own. The body can consist of multiple simple statements separated by semicolons, but this tends to be frowned upon.

In general, even though it's not always required, if you keep all your statements on individual lines and always indent your nested blocks, your code will be easier to read and change in the future. Moreover, some code profiling and coverage tools may not be able to distinguish between multiple statements squeezed onto a single line or the header and body of a one-line compound statement. It is almost always to your advantage to keep things simple in Python.

To see a prime and common exception to one of these rules in action, however (the use of a single-line `if` statement to break out of a loop), let's move on to the next section and write some real code.

A Quick Example: Interactive Loops

We'll see all these syntax rules in action when we tour Python's specific compound statements in the next few chapters, but they work the same everywhere in the Python language. To get started, let's work through a brief, realistic example that demonstrates the way that statement syntax and statement nesting come together in practice, and introduces a few statements along the way.

A Simple Interactive Loop

Suppose you're asked to write a Python program that interacts with a user in a console window. Maybe you're accepting inputs to send to a database, or reading numbers to be used in a calculation. Regardless of the purpose, you need to code a loop that reads one or more inputs from a user typing on a keyboard, and prints back a result for each. In other words, you need to write a classic read/evaluate/print loop program.

In Python, typical boilerplate code for such an interactive loop might look like this:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(reply.upper())
```

This code makes use of a few new ideas:

- The code leverages the Python `while` loop, Python's most general looping statement. We'll study the `while` statement in more detail later, but in short, it consists of the word `while`, followed by an expression that is interpreted as a true or false result, followed by a nested block of code that is repeated while the test at the top is true (the word `True` here is considered always true).
- The `input` built-in function we met earlier in the book is used here for general console input—it prints its optional argument string as a prompt and returns the user's typed reply as a string.
- A single-line `if` statement that makes use of the special rule for nested blocks also appears here: the body of the `if` appears on the header line after the colon instead

of being indented on a new line underneath it. This would work either way, but as it's coded, we've saved an extra line.

- Finally, the Python `break` statement is used to exit the loop immediately—it simply jumps out of the loop statement altogether, and the program continues after the loop. Without this exit statement, the `while` would loop forever, as its test is always true.

In effect, this combination of statements essentially means "read a line from the user and print it in uppercase until the user enters the word 'stop.'" There are other ways to code such a loop, but the form used here is very common in Python code.

Notice that all three lines nested under the `while` header line are indented the same amount—because they line up vertically in a column this way, they are the block of code that is associated with the `while` test and repeated. Either the end of the source file or a lesser-indented statement will terminate the loop body block.

When run, here is the sort of interaction we get from this code:

```
Enter text:spam
SPAM
Enter text:42
42
Enter text:stop
```

Version skew note: This example is coded for Python 3.0. If you are working in Python 2.6 or earlier, the code works the same, but you should use `raw_input` instead of `input`, and you can omit the outer parentheses in `print` statements. In 3.0 the former was renamed, and the latter is a built-in function instead of a statement (more on `print`s in the next chapter).

Doing Math on User Inputs

Our script works, but now suppose that instead of converting a text string to uppercase, we want to do some math with numeric input—squaring it, for example, perhaps in some misguided effort to discourage users who happen to be obsessed with youth. We might try statements like these to achieve the desired effect:

```
>>> reply = '20'
>>> reply ** 2
...error text omitted...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

This won't quite work in our script, though, because (as discussed in the prior part of the book) Python won't convert object types in expressions unless they are all numeric, and input from a user is always returned to our script as a string. We cannot raise a string of digits to a power unless we convert it manually to an integer:

```
>>> int(reply) ** 2
400
```

Armed with this information, we can now recode our loop to perform the necessary math. Type the following in a file to test it:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(int(reply) ** 2)
print('Bye')
```

This script uses a single-line `if` statement to exit on "stop" as before, but it also converts inputs to perform the required math. This version also adds an exit message at the bottom. Because the `print` statement in the last line is not indented as much as the nested block of code, it is not considered part of the loop body and will run only once, after the loop is exited:

```
Enter text:2
4
Enter text:40
1600
Enter text:stop
Bye
```

One note here: I'm assuming that this code is stored in and run from a script file. If you are entering this code interactively, be sure to include a blank line (i.e., press Enter twice) before the final `print` statement, to terminate the loop. The final `print` doesn't quite make sense in interactive mode, though (you'll have to code it after interacting with the loop!).

Handling Errors by Testing Inputs

So far so good, but notice what happens when the input is invalid:

```
Enter text:xxx
...error text omitted...
ValueError: invalid literal for int() with base 10: 'xxx'
```

The built-in `int` function raises an exception here in the face of a mistake. If we want our script to be robust, we can check the string's content ahead of time with the string object's `isdigit` method:

```
>>> S = '123'
>>> T = 'xxx'
>>> S.isdigit(), T.isdigit()
(True, False)
```

This also gives us an excuse to further nest the statements in our example. The following new version of our interactive script uses a full-blown `if` statement to work around the exception on errors:

```
while True:
    reply = input('Enter text:')
```

```
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
print('Bye')
```

We'll study the `if` statement in more detail in Chapter 12, but it's a fairly lightweight tool for coding logic in scripts. In its full form, it consists of the word `if` followed by a test and an associated block of code, one or more optional `elif` ("else if") tests and code blocks, and an optional `else` part, with an associated block of code at the bottom to serve as a default. Python runs the block of code associated with the first test that is true, working from top to bottom, or the `else` part if all tests are false.

The `if`, `elif`, and `else` parts in the preceding example are associated as part of the same statement because they all line up vertically (i.e., share the same level of indentation). The `if` statement spans from the word `if` to the start of the `print` statement on the last line of the script. In turn, the entire `if` block is part of the `while` loop because all of it is indented under the loop's header line. Statement nesting is natural once you get the hang of it.

When we run our new script, its code catches errors before they occur and prints an (arguably silly) error message to demonstrate:

```
Enter text:5
25
Enter text:xyz
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
Enter text:stop
```

Handling Errors with try Statements

The preceding solution works, but as you'll see later in the book, the most general way to handle errors in Python is to catch and recover from them completely using the Python `try` statement. We'll explore this statement in depth in Part VII of this book, but as a preview, using a `try` here can lead to code that some would claim is simpler than the prior version:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
print('Bye')
```

This version works exactly like the previous one, but we've replaced the explicit error check with code that assumes the conversion will work and wraps it up in an exception handler for cases when it doesn't. This `try` statement is composed of the word `try`, followed by the main block of code (the action we are trying to run), followed by an `except` part that gives the exception handler code and an `else` part to be run if no exception is raised in the `try` part. Python first runs the `try` part, then runs either the `except` part (if an exception occurs) or the `else` part (if no exception occurs).

In terms of statement nesting, because the words `try`, `except`, and `else` are all indented to the same level, they are all considered part of the same single `try` statement. Notice that the `else` part is associated with the `try` here, not the `if`. As we've seen, `else` can appear in `if` statements in Python, but it can also appear in `try` statements and loops—its indentation tells you what statement it is a part of. In this case, the `try` statement spans from the word `try` through the code indented under the word `else`, because the `else` is indented to the same level as `try`. The `if` statement in this code is a one-liner and ends after the `break`.

Again, we'll come back to the `try` statement later in this book. For now, be aware that because `try` can be used to intercept any error, it reduces the amount of error-checking code you have to write, and it's a very general approach to dealing with unusual cases. If we wanted to support input of floating-point numbers instead of just integers, for example, using `try` would be much easier than manual error testing—we could simply run a `float` call and catch its exceptions, instead of trying to analyze all possible floating-point syntax.

Nesting Code Three Levels Deep

Let's look at one last mutation of our script. Nesting can take us even further if we need it to—we could, for example, branch to one of a set of alternatives based on the relative magnitude of a valid input:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        num = int(reply)
        if num < 20:
            print('low')
        else:
            print(num ** 2)
print('Bye')
```

This version includes an `if` statement nested in the `else` clause of another `if` statement, which is in turn nested in the `while` loop. When code is conditional, or repeated like this, we simply indent it further to the right. The net effect is like that of the prior versions, but we'll now print "low" for numbers less than 20:

```
Enter text:19
low
Enter text:20
400
Enter text:spam
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

Chapter Summary

That concludes our quick look at Python statement syntax. This chapter introduced the general rules for coding statements and blocks of code. As you've learned, in Python we normally code one statement per line and indent all the statements in a nested block the same amount (indentation is part of Python's syntax). However, we also looked at a few exceptions to these rules, including continuation lines and single-line tests and loops. Finally, we put these ideas to work in an interactive script that demonstrated a handful of statements and showed statement syntax in action.

In the next chapter, we'll start to dig deeper by going over each of Python's basic procedural statements in depth. As you'll see, though, all statements follow the same general rules introduced here.

Test Your Knowledge: Quiz

1. What three things are required in a C-like language but omitted in Python?
2. How is a statement normally terminated in Python?
3. How are the statements in a nested block of code normally associated in Python?
4. How can you make a single statement span multiple lines?
5. How can you code a compound statement on a single line?
6. Is there any valid reason to type a semicolon at the end of a statement in Python?
7. What is a `try` statement for?
8. What is the most common coding mistake among Python beginners?

Test Your Knowledge: Answers

1. C-like languages require parentheses around the tests in some statements, semicolons at the end of each statement, and braces around a nested block of code.
2. The end of a line terminates the statement that appears on that line. Alternatively, if more than one statement appears on the same line, they can be terminated with semicolons; similarly, if a statement spans many lines, you must terminate it by closing a bracketed syntactic pair.
3. The statements in a nested block are all indented the same number of tabs or spaces.
4. A statement can be made to span many lines by enclosing part of it in parentheses, square brackets, or curly braces; the statement ends when Python sees a line that contains the closing part of the pair.
5. The body of a compound statement can be moved to the header line after the colon, but only if the body consists of only noncompound statements.
6. Only when you need to squeeze more than one statement onto a single line of code. Even then, this only works if all the statements are noncompound, and it's discouraged because it can lead to code that is difficult to read.
7. The `try` statement is used to catch and recover from exceptions (errors) in a Python script. It's usually an alternative to manually checking for errors in your code.
8. Forgetting to type the colon character at the end of the header line in a compound statement is the most common beginner's mistake. If you haven't made it yet, you probably will soon!

CHAPTER 11

Assignments, Expressions, and Prints

Now that we've had a quick introduction to Python statement syntax, this chapter begins our in-depth tour of specific Python statements. We'll begin with the basics: assignment statements, expression statements, and print operations. We've already seen all of these in action, but here we'll fill in important details we've skipped so far. Although they're fairly simple, as you'll see, there are optional variations for each of these statement types that will come in handy once you begin writing real Python programs.

Assignment Statements

We've been using the Python assignment statement for a while to assign objects to names. In its basic form, you write the *target* of an assignment on the left of an equals sign, and the *object* to be assigned on the right. The target on the left may be a name or object component, and the object on the right can be an arbitrary expression that computes an object. For the most part, assignments are straightforward, but here are a few properties to keep in mind:

- **Assignments create object references**. As discussed in Chapter 6, Python assignments store references to objects in names or data structure components. They always create references to objects instead of copying the objects. Because of that, Python variables are more like pointers than data storage areas.
- **Names are created when first assigned**. Python creates a variable name the first time you assign it a value (i.e., an object reference), so there's no need to predeclare names ahead of time. Some (but not all) data structure slots are created when assigned, too (e.g., dictionary entries, some object attributes). Once assigned, a name is replaced with the value it references whenever it appears in an expression.
- **Names must be assigned before being referenced**. It's an error to use a name to which you haven't yet assigned a value. Python raises an exception if you try, rather than returning some sort of ambiguous default value; if it returned a default instead, it would be more difficult for you to spot typos in your code.

- **Some operations perform assignments implicitly**. In this section we're concerned with the = statement, but assignment occurs in many contexts in Python. For instance, we'll see later that module imports, function and class definitions, `for` loop variables, and function arguments are all implicit assignments. Because assignment works the same everywhere it pops up, all these contexts simply bind names to object references at runtime.

Assignment Statement Forms

Although assignment is a general and pervasive concept in Python, we are primarily interested in assignment *statements* in this chapter. Table 11-1 illustrates the different assignment statement forms in Python.

Table 11-1. Assignment statement forms

Operation	Interpretation
`spam = 'Spam'`	Basic form
`spam, ham = 'yum', 'YUM'`	Tuple assignment (positional)
`[spam, ham] = ['yum', 'YUM']`	List assignment (positional)
`a, b, c, d = 'spam'`	Sequence assignment, generalized
`a, *b = 'spam'`	Extended sequence unpacking (Python 3.0)
`spam = ham = 'lunch'`	Multiple-target assignment
`spams += 42`	Augmented assignment (equivalent to `spams = spams + 42`)

The first form in Table 11-1 is by far the most common: binding a name (or data structure component) to a single object. In fact, you could get all your work done with this basic form alone. The other table entries represent special forms that are all optional, but that programmers often find convenient in practice:

Tuple- and list-unpacking assignments
: The second and third forms in the table are related. When you code a tuple or list on the left side of the =, Python pairs objects on the right side with targets on the left by position and assigns them from left to right. For example, in the second line of Table 11-1, the name `spam` is assigned the string `'yum'`, and the name `ham` is bound to the string `'YUM'`. In this case Python internally makes a tuple of the items on the right, which is why this is called tuple-unpacking assignment.

Sequence assignments
: In recent versions of Python, tuple and list assignments have been generalized into instances of what we now call *sequence assignment*—any sequence of names can be assigned to any sequence of values, and Python assigns the items one at a time by position. We can even mix and match the types of the sequences involved. The fourth line in Table 11-1, for example, pairs a tuple of names with a string of characters: `a` is assigned `'s'`, `b` is assigned `'p'`, and so on.

Extended sequence unpacking
: In Python 3.0, a new form of sequence assignment allows us to be more flexible in how we select portions of a sequence to assign. The fifth line in Table 11-1, for example, matches `a` with the first character in the string on the right and `b` with the rest: `a` is assigned `'s'`, and `b` is assigned `'pam'`. This provides a simpler alternative to assigning the results of manual slicing operations.

Multiple-target assignments
: The sixth line in Table 11-1 shows the multiple-target form of assignment. In this form, Python assigns a reference to the same object (the object farthest to the right) to all the targets on the left. In the table, the names `spam` and `ham` are both assigned references to the same string object, `'lunch'`. The effect is the same as if we had coded `ham = 'lunch'` followed by `spam = ham`, as `ham` evaluates to the original string object (i.e., not a separate copy of that object).

Augmented assignments
: The last line in Table 11-1 is an example of *augmented assignment*—a shorthand that combines an expression and an assignment in a concise way. Saying `spam += 42`, for example, has the same effect as `spam = spam + 42`, but the augmented form requires less typing and is generally quicker to run. In addition, if the subject is mutable and supports the operation, an augmented assignment may run even quicker by choosing an in-place update operation instead of an object copy. There is one augmented assignment statement for every binary expression operator in Python.

Sequence Assignments

We've already used basic assignments in this book. Here are a few simple examples of sequence-unpacking assignments in action:

```
% python
>>> nudge = 1
>>> wink  = 2
>>> A, B = nudge, wink              # Tuple assignment
>>> A, B                            # Like A = nudge; B = wink
(1, 2)
>>> [C, D] = [nudge, wink]          # List assignment
>>> C, D
(1, 2)
```

Notice that we really are coding two tuples in the third line in this interaction—we've just omitted their enclosing parentheses. Python pairs the values in the tuple on the right side of the assignment operator with the variables in the tuple on the left side and assigns the values one at a time.

Tuple assignment leads to a common coding trick in Python that was introduced in a solution to the exercises at the end of Part II. Because Python creates a temporary tuple that saves the original values of the variables on the right while the statement runs,

unpacking assignments are also a way to *swap* two variables' values without creating a temporary variable of your own—the tuple on the right remembers the prior values of the variables automatically:

```
>>> nudge = 1
>>> wink  = 2
>>> nudge, wink = wink, nudge       # Tuples: swaps values
>>> nudge, wink                     # Like T = nudge; nudge = wink; wink = T
(2, 1)
```

In fact, the original tuple and list assignment forms in Python have been generalized to accept any type of sequence on the right as long as it is of the same length as the sequence on the left. You can assign a tuple of values to a list of variables, a string of characters to a tuple of variables, and so on. In all cases, Python assigns items in the sequence on the right to variables in the sequence on the left by position, from left to right:

```
>>> [a, b, c] = (1, 2, 3)           # Assign tuple of values to list of names
>>> a, c
(1, 3)
>>> (a, b, c) = "ABC"               # Assign string of characters to tuple
>>> a, c
('A', 'C')
```

Technically speaking, sequence assignment actually supports any *iterable* object on the right, not just any sequence. This is a more general concept that we will explore in Chapters 14 and 20.

Advanced sequence assignment patterns

Although we can mix and match sequence types around the = symbol, we must have the *same number* of items on the right as we have variables on the left, or we'll get an error. Python 3.0 allows us to be more general with extended unpacking syntax, described in the next section. But normally, and always in Python 2.X, the number of items in the assignment target and subject must match:

```
>>> string = 'SPAM'
>>> a, b, c, d = string                          # Same number on both sides
>>> a, d
('S', 'M')

>>> a, b, c = string                             # Error if not
...error text omitted...
ValueError: too many values to unpack
```

To be more general, we can slice. There are a variety of ways to employ slicing to make this last case work:

```
>>> a, b, c = string[0], string[1], string[2:]     # Index and slice
>>> a, b, c
('S', 'P', 'AM')

>>> a, b, c = list(string[:2]) + [string[2:]]      # Slice and concatenate
>>> a, b, c
```

```
('S', 'P', 'AM')

>>> a, b = string[:2]                    # Same, but simpler
>>> c = string[2:]
>>> a, b, c
('S', 'P', 'AM')

>>> (a, b), c = string[:2], string[2:]   # Nested sequences
>>> a, b, c
('S', 'P', 'AM')
```

As the last example in this interaction demonstrates, we can even assign *nested* sequences, and Python unpacks their parts according to their shape, as expected. In this case, we are assigning a tuple of two items, where the first item is a nested sequence (a string), exactly as though we had coded it this way:

```
>>> ((a, b), c) = ('SP', 'AM')           # Paired by shape and position
>>> a, b, c
('S', 'P', 'AM')
```

Python pairs the first string on the right (`'SP'`) with the first tuple on the left (`(a, b)`) and assigns one character at a time, before assigning the entire second string (`'AM'`) to the variable `c` all at once. In this event, the sequence-nesting shape of the object on the left must match that of the object on the right. Nested sequence assignment like this is somewhat advanced, and rare to see, but it can be convenient for picking out the parts of data structures with known shapes.

For example, we'll see in Chapter 13 that this technique also works in `for` loops, because loop items are assigned to the target given in the loop header:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ...          # Simple tuple assignment

for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ...    # Nested tuple assignment
```

In a note in Chapter 18, we'll also see that this nested tuple (really, sequence) unpacking assignment form works for function argument lists in Python 2.6 (though not in 3.0), because function arguments are passed by assignment as well:

```
def f(((a, b), c)):              # For arguments too in Python 2.6, but not 3.0
f(((1, 2), 3))
```

Sequence-unpacking assignments also give rise to another common coding idiom in Python—assigning an integer series to a set of variables:

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
```

This initializes the three names to the integer codes `0`, `1`, and `2`, respectively (it's Python's equivalent of the *enumerated* data types you may have seen in other languages). To make sense of this, you need to know that the `range` built-in function generates a list of successive integers:

```
>>> range(3)                          # Use list(range(3)) in Python 3.0
[0, 1, 2]
```

Because `range` is commonly used in `for` loops, we'll say more about it in Chapter 13.

Another place you may see a tuple assignment at work is for splitting a sequence into its front and the rest in loops like this:

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, L = L[0], L[1:]          # See next section for 3.0 alternative
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

The tuple assignment in the loop here could be coded as the following two lines instead, but it's often more convenient to string them together:

```
...     front = L[0]
...     L = L[1:]
```

Notice that this code is using the list as a sort of stack data structure, which can often also be achieved with the `append` and `pop` methods of list objects; here, `front = L.pop(0)` would have much the same effect as the tuple assignment statement, but it would be an in-place change. We'll learn more about `while` loops, and other (often better) ways to step through a sequence with `for` loops, in Chapter 13.

Extended Sequence Unpacking in Python 3.0

The prior section demonstrated how to use manual slicing to make sequence assignments more general. In Python 3.0 (but not 2.6), sequence assignment has been generalized to make this easier. In short, a single *starred name*, `*X`, can be used in the assignment target in order to specify a more general matching against the sequence—the starred name is assigned a list, which collects all items in the sequence not assigned to other names. This is especially handy for common coding patterns such as splitting a sequence into its "front" and "rest", as in the preceding section's last example.

Extended unpacking in action

Let's look at an example. As we've seen, sequence assignments normally require exactly as many names in the target on the left as there are items in the subject on the right. We get an error if the lengths disagree (unless we manually sliced on the right, as shown in the prior section):

```
C:\misc> c:\python30\python
>>> seq = [1, 2, 3, 4]
>>> a, b, c, d = seq
>>> print(a, b, c, d)
1 2 3 4
```

```
>>> a, b = seq
ValueError: too many values to unpack
```

In Python 3.0, though, we can use a single starred name in the target to match more generally. In the following continuation of our interactive session, `a` matches the first item in the sequence, and `b` matches the rest:

```
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

When a starred name is used, the number of items in the target on the left need not match the length of the subject sequence. In fact, the starred name can appear anywhere in the target. For instance, in the next interaction `b` matches the last item in the sequence, and `a` matches everything before the last:

```
>>> *a, b = seq
>>> a
[1, 2, 3]
>>> b
4
```

When the starred name appears in the middle, it collects everything between the other names listed. Thus, in the following interaction `a` and `c` are assigned the first and last items, and `b` gets everything in between them:

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
4
```

More generally, wherever the starred name shows up, it will be assigned a list that collects every unassigned name at that position:

```
>>> a, b, *c = seq
>>> a
1
>>> b
2
>>> c
[3, 4]
```

Naturally, like normal sequence assignment, extended sequence unpacking syntax works for any sequence types, not just lists. Here it is unpacking characters in a string:

```
>>> a, *b = 'spam'
>>> a, b
('s', ['p', 'a', 'm'])

>>> a, *b, c = 'spam'
```

```
>>> a, b, c
('s', ['p', 'a'], 'm')
```

This is similar in spirit to slicing, but not exactly the same—a sequence unpacking assignment always returns a *list* for multiple matched items, whereas slicing returns a sequence of the same type as the object sliced:

```
>>> S = 'spam'

>>> S[0], S[1:]      # Slices are type-specific, * assignment always returns a list
('s', 'pam')

>>> S[0], S[1:3], S[3]
('s', 'pa', 'm')
```

Given this extension in 3.0, as long as we're processing a list the last example of the prior section becomes even simpler, since we don't have to manually slice to get the first and rest of the items:

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, *L = L                    # Get first, rest without slicing
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

Boundary cases

Although extended sequence unpacking is flexible, some boundary cases are worth noting. First, the starred name may match just a single item, but is always assigned a list:

```
>>> seq
[1, 2, 3, 4]

>>> a, b, c, *d = seq
>>> print(a, b, c, d)
1 2 3 [4]
```

Second, if there is nothing left to match the starred name, it is assigned an empty list, regardless of where it appears. In the following, `a`, `b`, `c`, and `d` have matched every item in the sequence, but Python assigns `e` an empty list instead of treating this as an error case:

```
>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []

>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

Finally, errors can still be triggered if there is more than one starred name, if there are too few values and no star (as before), and if the starred name is not itself coded inside a sequence:

```
>>> a, *b, c, *d = seq
SyntaxError: two starred expressions in assignment

>>> a, b = seq
ValueError: too many values to unpack

>>> *a = seq
SyntaxError: starred assignment target must be in a list or tuple

>>> *a, = seq
>>> a
[1, 2, 3, 4]
```

A useful convenience

Keep in mind that extended sequence unpacking assignment is just a convenience. We can usually achieve the same effects with explicit indexing and slicing (and in fact must in Python 2.X), but extended unpacking is simpler to code. The common "first, rest" splitting coding pattern, for example, can be coded either way, but slicing involves extra work:

```
>>> seq
[1, 2, 3, 4]

>>> a, *b = seq                         # First, rest
>>> a, b
(1, [2, 3, 4])

>>> a, b = seq[0], seq[1:]              # First, rest: traditional
>>> a, b
(1, [2, 3, 4])
```

The also common "rest, last" splitting pattern can similarly be coded either way, but the new extended unpacking syntax requires noticeably fewer keystrokes:

```
>>> *a, b = seq                         # Rest, last
>>> a, b
([1, 2, 3], 4)

>>> a, b = seq[:-1], seq[-1]            # Rest, last: traditional
>>> a, b
([1, 2, 3], 4)
```

Because it is not only simpler but, arguably, more natural, extended sequence unpacking syntax will likely become widespread in Python code over time.

Application to for loops

Because the loop variable in the `for` loop statement can be any assignment target, extended sequence assignment works here too. We met the `for` loop iteration tool briefly in Part II and will study it formally in Chapter 13. In Python 3.0, extended assignments may show up after the word `for`, where a simple variable name is more commonly used:

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    ...
```

When used in this context, on each iteration Python simply assigns the next tuple of values to the tuple of names. On the first loop, for example, it's as if we'd run the following assignment statement:

```
a, *b, c = (1, 2, 3, 4)                          # b gets [2, 3]
```

The names `a`, `b`, and `c` can be used within the loop's code to reference the extracted components. In fact, this is really not a special case at all, but just an instance of general assignment at work. As we saw earlier in this chapter, we can do the same thing with simple tuple assignment in both Python 2.X and 3.X:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:          # a, b, c = (1, 2, 3), ...
```

And we can always emulate 3.0's extended assignment behavior in 2.6 by manually slicing:

```
for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    a, b, c = all[0], all[1:3], all[3]
```

Since we haven't learned enough to get more detailed about the syntax of `for` loops, we'll return to this topic in Chapter 13.

Multiple-Target Assignments

A multiple-target assignment simply assigns all the given names to the object all the way to the right. The following, for example, assigns the three variables `a`, `b`, and `c` to the string `'spam'`:

```
>>> a = b = c = 'spam'
>>> a, b, c
('spam', 'spam', 'spam')
```

This form is equivalent to (but easier to code than) these three assignments:

```
>>> c = 'spam'
>>> b = c
>>> a = b
```

Multiple-target assignment and shared references

Keep in mind that there is just one object here, shared by all three variables (they all wind up pointing to the same object in memory). This behavior is fine for immutable types—for example, when initializing a set of counters to zero (recall that variables

must be assigned before they can be used in Python, so you must initialize counters to zero before you can start adding to them):

```
>>> a = b = 0
>>> b = b + 1
>>> a, b
(0, 1)
```

Here, changing `b` only changes `b` because numbers do not support in-place changes. As long as the object assigned is immutable, it's irrelevant if more than one name references it.

As usual, though, we have to be more cautious when initializing variables to an empty mutable object such as a list or dictionary:

```
>>> a = b = []
>>> b.append(42)
>>> a, b
([42], [42])
```

This time, because `a` and `b` reference the same object, appending to it in-place through `b` will impact what we see through `a` as well. This is really just another example of the shared reference phenomenon we first met in Chapter 6. To avoid the issue, initialize mutable objects in separate statements instead, so that each creates a distinct empty object by running a distinct literal expression:

```
>>> a = []
>>> b = []
>>> b.append(42)
>>> a, b
([], [42])
```

Augmented Assignments

Beginning with Python 2.0, the set of additional assignment statement formats listed in Table 11-2 became available. Known as *augmented assignments*, and borrowed from the C language, these formats are mostly just shorthand. They imply the combination of a binary expression and an assignment. For instance, the following two formats are now roughly equivalent:

```
X = X + Y                     # Traditional form
X += Y                        # Newer augmented form
```

Table 11-2. Augmented assignment statements

`X += Y`	`X &= Y`	`X -= Y`	`X |= Y`
`X *= Y`	`X ^= Y`	`X /= Y`	`X >>= Y`
`X %= Y`	`X <<= Y`	`X **= Y`	`X //= Y`

Augmented assignment works on any type that supports the implied binary expression. For example, here are two ways to add 1 to a name:

```
>>> x = 1
>>> x = x + 1                    # Traditional
>>> x
2
>>> x += 1                       # Augmented
>>> x
3
```

When applied to a string, the augmented form performs concatenation instead. Thus, the second line here is equivalent to typing the longer `S = S + "SPAM"`:

```
>>> S = "spam"
>>> S += "SPAM"                  # Implied concatenation
>>> S
'spamSPAM'
```

As shown in Table 11-2, there are analogous augmented assignment forms for every Python binary expression operator (i.e., each operator with values on the left and right side). For instance, `X *= Y` multiplies and assigns, `X >>= Y` shifts right and assigns, and so on. `X //= Y` (for floor division) was added in version 2.2.

Augmented assignments have three advantages:*

- There's less for you to type. Need I say more?
- The left side only has to be evaluated once. In `X += Y`, `X` may be a complicated object expression. In the augmented form, it only has to be evaluated once. However, in the long form, `X = X + Y`, `X` appears twice and must be run twice. Because of this, augmented assignments usually run faster.
- The optimal technique is automatically chosen. That is, for objects that support in-place changes, the augmented forms automatically perform in-place change operations instead of slower copies.

The last point here requires a bit more explanation. For augmented assignments, in-place operations may be applied for mutable objects as an optimization. Recall that lists can be extended in a variety of ways. To add a single item to the end of a list, we can concatenate or call `append`:

```
>>> L = [1, 2]
>>> L = L + [3]                  # Concatenate: slower
>>> L
[1, 2, 3]
>>> L.append(4)                  # Faster, but in-place
>>> L
[1, 2, 3, 4]
```

* C/C++ programmers take note: although Python now supports statements like `X += Y`, it still does not have C's auto-increment/decrement operators (e.g., `X++`, `--X`). These don't quite map to the Python object model because Python has no notion of in-place changes to immutable objects like numbers.

And to add a set of items to the end, we can either concatenate again or call the list `extend` method:†

```
>>> L = L + [5, 6]              # Concatenate: slower
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8])            # Faster, but in-place
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

In both cases, concatenation is less prone to the side effects of shared object references but will generally run slower than the in-place equivalent. Concatenation operations must create a new object, copy in the list on the left, and then copy in the list on the right. By contrast, in-place method calls simply add items at the end of a memory block.

When we use augmented assignment to extend a list, we can forget these details—for example, Python automatically calls the quicker `extend` method instead of using the slower concatenation operation implied by `+`:

```
>>> L += [9, 10]                # Mapped to L.extend([9, 10])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Augmented assignment and shared references

This behavior is usually what we want, but notice that it implies that the `+=` is an in-place change for lists; thus, it is not exactly like `+` concatenation, which always makes a new object. As for all shared reference cases, this difference might matter if other names reference the object being changed:

```
>>> L = [1, 2]
>>> M = L                       # L and M reference the same object
>>> L = L + [3, 4]              # Concatenation makes a new object
>>> L, M                        # Changes L but not M
([1, 2, 3, 4], [1, 2])

>>> L = [1, 2]
>>> M = L
>>> L += [3, 4]                 # But += really means extend
>>> L, M                        # M sees the in-place change too!
([1, 2, 3, 4], [1, 2, 3, 4])
```

This only matters for mutables like lists and dictionaries, and it is a fairly obscure case (at least, until it impacts your code!). As always, make copies of your mutable objects if you need to break the shared reference structure.

† As suggested in Chapter 6, we can also use slice assignment (e.g., `L[len(L):] = [11,12,13]`), but this works roughly the same as the simpler list `extend` method.

Variable Name Rules

Now that we've explored assignment statements, it's time to get more formal about the use of variable names. In Python, names come into existence when you assign values to them, but there are a few rules to follow when picking names for things in your programs:

Syntax: (underscore or letter) + (any number of letters, digits, or underscores)
: Variable names must start with an underscore or letter, which can be followed by any number of letters, digits, or underscores. `_spam`, `spam`, and `Spam_1` are legal names, but `1_Spam`, `spam$`, and `@#!` are not.

Case matters: `SPAM` *is not the same as* `spam`
: Python always pays attention to case in programs, both in names you create and in reserved words. For instance, the names `X` and `x` refer to two different variables. For portability, case also matters in the names of imported module files, even on platforms where the filesystems are case-insensitive.

Reserved words are off-limits
: Names you define cannot be the same as words that mean special things in the Python language. For instance, if you try to use a variable name like `class`, Python will raise a syntax error, but `klass` and `Class` work fine. Table 11-3 lists the words that are currently reserved (and hence off-limits for names of your own) in Python.

Table 11-3. Python 3.0 reserved words

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	
`break`	`except`	`in`	`raise`	

Table 11-3 is specific to Python 3.0. In Python 2.6, the set of reserved words differs slightly:

- `print` is a reserved word, because printing is a statement, not a built-in (more on this later in this chapter).
- `exec` is a reserved word, because it is a statement, not a built-in function.
- `nonlocal` is not a reserved word because this statement is not available.

In older Pythons the story is also more or less the same, with a few variations:

- `with` and `as` were not reserved until 2.6, when context managers were officially enabled.
- `yield` was not reserved until Python 2.3, when generator functions were enabled.
- `yield` morphed from statement to expression in 2.5, but it's still a reserved word, not a built-in function.

As you can see, most of Python's reserved words are all lowercase. They are also all truly reserved—unlike names in the built-in scope that you will meet in the next part of this book, you cannot redefine reserved words by assignment (e.g., `and = 1` results in a syntax error).‡

Besides being of mixed case, the first three entries in Table 11-3, `True`, `False`, and `None`, are somewhat unusual in meaning—they also appear in the built-in scope of Python described in Chapter 17, and they are technically names assigned to objects. They are truly reserved in all other senses, though, and cannot be used for any other purpose in your script other than that of the objects they represent. All the other reserved words are hardwired into Python's syntax and can appear only in the specific contexts for which they are intended.

Furthermore, because module names in `import` statements become variables in your scripts, variable name constraints extend to your *module filenames* too. For instance, you can code files called *and.py* and *my-code.py* and run them as top-level scripts, but you cannot import them: their names without the ".py" extension become *variables* in your code and so must follow all the variable rules just outlined. Reserved words are off-limits, and dashes won't work, though underscores will. We'll revisit this idea in Part V of this book.

Python's Deprecation Protocol

It is interesting to note how reserved word changes are gradually phased into the language. When a new feature might break existing code, Python normally makes it an option and begins issuing "deprecation" warnings one or more releases before the feature is officially enabled. The idea is that you should have ample time to notice the warnings and update your code before migrating to the new release. This is not true for major new releases like 3.0 (which breaks existing code freely), but it is generally true in other cases.

For example, `yield` was an optional extension in Python 2.2, but is a standard keyword as of 2.3. It is used in conjunction with generator functions. This was one of a small handful of instances where Python broke with backward compatibility. Still, `yield` was phased in over time: it began generating deprecation warnings in 2.2 and was not enabled until 2.3.

‡ In the Jython Java-based implementation of Python, though, user-defined variable names can sometimes be the same as Python reserved words. See Chapter 2 for an overview of the Jython system.

Similarly, in Python 2.6, the words `with` and `as` become new reserved words for use in context managers (a newer form of exception handling). These two words are not reserved in 2.5, unless the context manager feature is turned on manually with a `from__future__import` (discussed later in this book). When used in 2.5, `with` and `as` generate warnings about the upcoming change—except in the version of IDLE in Python 2.5, which appears to have enabled this feature for you (that is, using these words as variable names does generate errors in 2.5, but only in its version of the IDLE GUI).

Naming conventions

Besides these rules, there is also a set of naming *conventions*—rules that are not required but are followed in normal practice. For instance, because names with two leading and trailing underscores (e.g., `__name__`) generally have special meaning to the Python interpreter, you should avoid this pattern for your own names. Here is a list of the conventions Python follows:

- Names that begin with a single underscore (`_X`) are not imported by a `from module import *` statement (described in Chapter 22).
- Names that have two leading and trailing underscores (`__X__`) are system-defined names that have special meaning to the interpreter.
- Names that begin with two underscores and do not end with two more (`__X`) are localized ("mangled") to enclosing classes (see the discussion of pseudoprivate attributes in Chapter 30).
- The name that is just a single underscore (`_`) retains the result of the last expression when working interactively.

In addition to these Python interpreter conventions, there are various other conventions that Python programmers usually follow. For instance, later in the book we'll see that class names commonly start with an uppercase letter and module names with a lowercase letter, and that the name `self`, though not reserved, usually has a special role in classes. In Chapter 17 we'll also study another, larger category of names known as the *built-ins*, which are predefined but not reserved (and so can be reassigned: `open = 42` works, though sometimes you might wish it didn't!).

Names have no type, but objects do

This is mostly review, but remember that it's crucial to keep Python's distinction between names and objects clear. As described in Chapter 6, objects have a type (e.g., integer, list) and may be mutable or not. Names (a.k.a. variables), on the other hand, are always just references to objects; they have no notion of mutability and have no associated type information, apart from the type of the object they happen to reference at a given point in time.

Thus, it's OK to assign the same name to different kinds of objects at different times:

```
>>> x = 0                   # x bound to an integer object
>>> x = "Hello"             # Now it's a string
>>> x = [1, 2, 3]           # And now it's a list
```

In later examples, you'll see that this generic nature of names can be a decided advantage in Python programming. In Chapter 17, you'll also learn that names also live in something called a *scope*, which defines where they can be used; the place where you assign a name determines where it is visible.§

For additional naming suggestions, see the previous section "Naming conventions" of Python's semi-official style guide, known as *PEP 8*. This guide is available at *http://www.python.org/dev/peps/pep-0008*, or via a web search for "Python PEP 8." Technically, this document formalizes coding standards for Python library code.

Though useful, the usual caveats about coding standards apply here. For one thing, PEP 8 comes with more detail than you are probably ready for at this point in the book. And frankly, it has become more complex, rigid, and subjective than it needs to be—some of its suggestions are not at all universally accepted or followed by Python programmers doing real work. Moreover, some of the most prominent companies using Python today have adopted coding standards of their own that differ.

PEP 8 does codify useful rule-of-thumb Python knowledge, though, and it's a great read for Python beginners, as long as you take its recommendations as guidelines, not gospel.

Expression Statements

In Python, you can use an expression as a statement, too—that is, on a line by itself. But because the result of the expression won't be saved, it usually makes sense to do so only if the expression does something useful as a side effect. Expressions are commonly used as statements in two situations:

For calls to functions and methods
: Some functions and methods do lots of work without returning a value. Such functions are sometimes called *procedures* in other languages. Because they don't return values that you might be interested in retaining, you can call these functions with expression statements.

§ If you've used a more restrictive language like C++, you may be interested to know that there is no notion of C++'s `const` declaration in Python; certain objects may be *immutable*, but names can always be assigned. Python also has ways to hide names in classes and modules, but they're not the same as C++'s declarations (if hiding attributes matters to you, see the coverage of `_X` module names in Chapter 24, `__X` class names in Chapter 30, and the Private and Public class decorators example in Chapter 38).

For printing values at the interactive prompt
: Python echoes back the results of expressions typed at the interactive command line. Technically, these are expression statements, too; they serve as a shorthand for typing `print` statements.

Table 11-4 lists some common expression statement forms in Python. Calls to functions and methods are coded with zero or more argument objects (really, expressions that evaluate to objects) in parentheses, after the function/method name.

Table 11-4. Common Python expression statements

Operation	Interpretation
`spam(eggs, ham)`	Function calls
`spam.ham(eggs)`	Method calls
`spam`	Printing variables in the interactive interpreter
`print(a, b, c, sep='')`	Printing operations in Python 3.0
`yield x ** 2`	Yielding expression statements

The last two entries in Table 11-4 are somewhat special cases—as we'll see later in this chapter, printing in Python 3.0 is a function call usually coded on a line by itself, and the `yield` operation in generator functions (discussed in Chapter 20) is often coded as a statement as well. Both are really just instances of expression statements.

For instance, though you normally run a `print` call on a line by itself as an expression statement, it returns a value like any other function call (its return value is `None`, the default return value for functions that don't return anything meaningful):

```
>>> x = print('spam')          # print is a function call expression in 3.0
spam
>>> print(x)                   # But it is coded as an expression statement
None
```

Also keep in mind that although expressions can appear as statements in Python, statements cannot be used as expressions. For example, Python doesn't allow you to embed assignment statements (`=`) in other expressions. The rationale for this is that it avoids common coding mistakes; you can't accidentally change a variable by typing `=` when you really mean to use the `==` equality test. You'll see how to code around this when you meet the Python `while` loop in Chapter 13.

Expression Statements and In-Place Changes

This brings up a mistake that is common in Python work. Expression statements are often used to run list methods that change a list in-place:

```
>>> L = [1, 2]
>>> L.append(3)                # Append is an in-place change
>>> L
[1, 2, 3]
```

However, it's not unusual for Python newcomers to code such an operation as an assignment statement instead, intending to assign `L` to the larger list:

```
>>> L = L.append(4)          # But append returns None, not L
>>> print(L)                 # So we lose our list!
None
```

This doesn't quite work, though. Calling an in-place change operation such as `append`, `sort`, or `reverse` on a list always changes the list in-place, but these methods do not return the list they have changed; instead, they return the `None` object. Thus, if you assign such an operation's result back to the variable name, you effectively lose the list (and it is probably garbage collected in the process!).

The moral of the story is, don't do this. We'll revisit this phenomenon in the section "Common Coding Gotchas" on page 387 at the end of this part of the book because it can also appear in the context of some looping statements we'll meet in later chapters.

Print Operations

In Python, `print` prints things—it's simply a programmer-friendly interface to the standard output stream.

Technically, printing converts one or more objects to their textual representations, adds some minor formatting, and sends the resulting text to either standard output or another file-like stream. In a bit more detail, `print` is strongly bound up with the notions of files and streams in Python:

File object methods
: In Chapter 9, we learned about file object methods that write text (e.g., `file.write(str)`). Printing operations are similar, but more focused—whereas file write methods write strings to arbitrary files, `print` writes objects to the `stdout` stream by default, with some automatic formatting added. Unlike with file methods, there is no need to convert objects to strings when using print operations.

Standard output stream
: The standard output stream (often known as `stdout`) is simply a default place to send a program's text output. Along with the standard input and error streams, it's one of three data connections created when your script starts. The standard output stream is usually mapped to the window where you started your Python program, unless it's been redirected to a file or pipe in your operating system's shell.

  Because the standard output stream is available in Python as the `stdout` file object in the built-in `sys` module (i.e., `sys.stdout`), it's possible to emulate `print` with file write method calls. However, `print` is noticeably easier to use and makes it easy to print text to other files and streams.

Printing is also one of the most visible places where Python 3.0 and 2.6 have diverged. In fact, this divergence is usually the first reason that most 2.X code won't run unchanged under 3.X. Specifically, the way you code print operations depends on which version of Python you use:

- In Python 3.X, printing is a *built-in function*, with keyword arguments for special modes.
- In Python 2.X, printing is a *statement* with specific syntax all its own.

Because this book covers both 3.0 and 2.6, we will look at each form in turn here. If you are fortunate enough to be able to work with code written for just one version of Python, feel free to pick the section that is relevant to you; however, as your circumstances may change, it probably won't hurt to be familiar with both cases.

The Python 3.0 print Function

Strictly speaking, printing is not a separate statement form in 3.0. Instead, it is simply an instance of the *expression statement* we studied in the preceding section.

The `print` built-in function is normally called on a line of its own, because it doesn't return any value we care about (technically, it returns `None`). Because it is a normal function, though, printing in 3.0 uses *standard function-call syntax*, rather than a special statement form. Because it provides special operation modes with keyword arguments, this form is both more general and supports future enhancements better.

By comparison, Python 2.6 `print` statements have somewhat ad-hoc syntax to support extensions such as end-of-line suppression and target files. Further, the 2.6 statement does not support separator specification at all; in 2.6, you wind up building strings ahead of time more often than you do in 3.0.

Call format

Syntactically, calls to the 3.0 `print` function have the following form:

```
print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout])
```

In this formal notation, items in square brackets are optional and may be omitted in a given call, and values after `=` give argument defaults. In English, this built-in function prints the textual representation of one or more `object`s separated by the string `sep` and followed by the string `end` to the stream `file`.

The `sep`, `end`, and `file` parts, if present, must be given as *keyword arguments*—that is, you must use a special "name=value" syntax to pass the arguments by name instead of position. Keyword arguments are covered in depth in Chapter 18, but they're straightforward to use. The keyword arguments sent to this call may appear in any left-to-right order following the objects to be printed, and they control the `print` operation:

- `sep` is a string inserted between each object's text, which defaults to a single space if not passed; passing an empty string suppresses separators altogether.
- `end` is a string added at the end of the printed text, which defaults to a `\n` newline character if not passed. Passing an empty string avoids dropping down to the next output line at the end of the printed text—the next `print` will keep adding to the end of the current output line.
- `file` specifies the file, standard stream, or other file-like object to which the text will be sent; it defaults to the `sys.stdout` standard output stream if not passed. Any object with a file-like `write(string)` method may be passed, but real files should be already opened for output.

The textual representation of each `object` to be printed is obtained by passing the object to the `str` built-in call; as we've seen, this built-in returns a "user friendly" display string for any object.‖ With no arguments at all, the `print` function simply prints a newline character to the standard output stream, which usually displays a blank line.

The 3.0 print function in action

Printing in 3.0 is probably simpler than some of its details may imply. To illustrate, let's run some quick examples. The following prints a variety of object types to the default standard output stream, with the default separator and end-of-line formatting added (these are the defaults because they are the most common use case):

```
C:\misc> c:\python30\python
>>>
>>> print()                                  # Display a blank line

>>> x = 'spam'
>>> y = 99
>>> z = ['eggs']
>>>
>>> print(x, y, z)                           # Print 3 objects per defaults
spam 99 ['eggs']
```

There's no need to convert objects to strings here, as would be required for file write methods. By default, `print` calls add a space between the objects printed. To suppress this, send an empty string to the `sep` keyword argument, or send an alternative separator of your choosing:

```
>>> print(x, y, z, sep='')                   # Suppress separator
spam99['eggs']
>>>
>>> print(x, y, z, sep=', ')                 # Custom separator
spam, 99, ['eggs']
```

‖ Technically, printing uses the equivalent of `str` in the internal implementation of Python, but the effect is the same. Besides this to-string conversion role, `str` is also the name of the string data type and can be used to decode Unicode strings from raw bytes with an extra encoding argument, as we'll learn in Chapter 36; this latter role is an advanced usage that we can safely ignore here.

Also by default, `print` adds an end-of-line character to terminate the output line. You can suppress this and avoid the line break altogether by passing an empty string to the `end` keyword argument, or you can pass a different terminator of your own (include a `\n` character to break the line manually):

```
>>> print(x, y, z, end='')                    # Suppress line break
spam 99 ['eggs']>>>
>>>
>>> print(x, y, z, end=''); print(x, y, z)    # Two prints, same output line
spam 99 ['eggs']spam 99 ['eggs']
>>> print(x, y, z, end='...\n')               # Custom line end
spam 99 ['eggs']...
>>>
```

You can also combine keyword arguments to specify both separators and end-of-line strings—they may appear in any order but must appear after all the objects being printed:

```
>>> print(x, y, z, sep='...', end='!\n')      # Multiple keywords
spam...99...['eggs']!
>>> print(x, y, z, end='!\n', sep='...')      # Order doesn't matter
spam...99...['eggs']!
```

Here is how the `file` keyword argument is used—it directs the printed text to an open output file or other compatible object for the duration of the single `print` (this is really a form of stream redirection, a topic we will revisit later in this section):

```
>>> print(x, y, z, sep='...', file=open('data.txt', 'w'))    # Print to a file
>>> print(x, y, z)                                           # Back to stdout
spam 99 ['eggs']
>>> print(open('data.txt').read())                           # Display file text
spam...99...['eggs']
```

Finally, keep in mind that the separator and end-of-line options provided by print operations are just conveniences. If you need to display more specific formatting, don't print this way, Instead, build up a more complex string ahead of time or within the `print` itself using the string tools we met in Chapter 7, and print the string all at once:

```
>>> text = '%s: %-.4f, %05d' % ('Result', 3.14159, 42)
>>> print(text)
Result: 3.1416, 00042
>>> print('%s: %-.4f, %05d' % ('Result', 3.14159, 42))
Result: 3.1416, 00042
```

As we'll see in the next section, almost everything we've just seen about the 3.0 `print` function also applies directly to 2.6 `print` statements—which makes sense, given that the function was intended to both emulate and improve upon 2.6 printing support.

The Python 2.6 print Statement

As mentioned earlier, printing in Python 2.6 uses a statement with unique and specific syntax, rather than a built-in function. In practice, though, 2.6 printing is mostly a variation on a theme; with the exception of separator strings (which are supported in

3.0 but not 2.6), everything we can do with the 3.0 `print` function has a direct translation to the 2.6 `print` statement.

Statement forms

Table 11-5 lists the `print` statement's forms in Python 2.6 and gives their Python 3.0 `print` function equivalents for reference. Notice that the *comma* is significant in `print` statements—it separates objects to be printed, and a trailing comma suppresses the end-of-line character normally added at the end of the printed text (not to be confused with tuple syntax!). The `>>` syntax, normally used as a bitwise right-shift operation, is used here as well, to specify a target output stream other than the `sys.stdout` default.

Table 11-5. Python 2.6 print statement forms

Python 2.6 statement	Python 3.0 equivalent	Interpretation
`print x, y`	`print(x, y)`	Print objects' textual forms to `sys.stdout`; add a space between the items and an end-of-line at the end
`print x, y,`	`print(x, y, end='')`	Same, but don't add end-of-line at end of text
`print >> afile, x, y`	`print(x, y, file=afile)`	Send text to `myfile.write`, not to `sys.stdout.write`

The 2.6 print statement in action

Although the 2.6 `print` statement has more unique syntax than the 3.0 function, it's similarly easy to use. Let's turn to some basic examples again. By default, the 2.6 `print` statement adds a space between the items separated by commas and adds a line break at the end of the current output line:

```
C:\misc> c:\python26\python
>>>
>>> x = 'a'
>>> y = 'b'
>>> print x, y
a b
```

This formatting is just a default; you can choose to use it or not. To suppress the line break so you can add more text to the current line later, end your `print` statement with a comma, as shown in the second line of Table 11-5 (the following is two statements on one line, separated by a semicolon):

```
>>> print x, y,; print x, y
a b a b
```

To suppress the space between items, again, don't print this way. Instead, build up an output string using the string concatenation and formatting tools covered in Chapter 7, and print the string all at once:

```
>>> print x + y
ab
>>> print '%s...%s' % (x, y)
a...b
```

As you can see, apart from their special syntax for usage modes, 2.6 `print` statements are roughly as simple to use as 3.0's function. The next section uncovers the way that files are specified in 2.6 `print`s.

Print Stream Redirection

In both Python 3.0 and 2.6, printing sends text to the standard output stream by default. However, it's often useful to send it elsewhere—to a text file, for example, to save results for later use or testing purposes. Although such redirection can be accomplished in system shells outside Python itself, it turns out to be just as easy to redirect a script's streams from within the script.

The Python "hello world" program

Let's start off with the usual (and largely pointless) language benchmark—the "hello world" program. To print a "hello world" message in Python, simply print the string per your version's print operation:

```
>>> print('hello world')                # Print a string object in 3.0
hello world

>>> print 'hello world'                 # Print a string object in 2.6
hello world
```

Because expression results are echoed on the interactive command line, you often don't even need to use a `print` statement there—simply type the expressions you'd like to have printed, and their results are echoed back:

```
>>> 'hello world'                       # Interactive echoes
'hello world'
```

This code isn't exactly an earth-shattering piece of software mastery, but it serves to illustrate printing behavior. Really, the `print` operation is just an ergonomic feature of Python—it provides a simple interface to the `sys.stdout` object, with a bit of default formatting. In fact, if you enjoy working harder than you must, you can also code print operations this way:

```
>>> import sys                          # Printing the hard way
>>> sys.stdout.write('hello world\n')
hello world
```

This code explicitly calls the `write` method of `sys.stdout`—an attribute preset when Python starts up to an open file object connected to the output stream. The `print` operation hides most of those details, providing a simple tool for simple printing tasks.

Manual stream redirection

So, why did I just show you the hard way to print? The `sys.stdout` print equivalent turns out to be the basis of a common technique in Python. In general, `print` and `sys.stdout` are directly related as follows. This statement:

```
print(X, Y)                              # Or, in 2.6: print X, Y
```

is equivalent to the longer:

```
import sys
sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
```

which manually performs a string conversion with `str`, adds a separator and newline with `+`, and calls the output stream's `write` method. Which would you rather code? (He says, hoping to underscore the programmer-friendly nature of prints....)

Obviously, the long form isn't all that useful for printing by itself. However, it is useful to know that this is exactly what `print` operations do because it is possible to *reassign* `sys.stdout` to something different from the standard output stream. In other words, this equivalence provides a way of making your `print` operations send their text to other places. For example:

```
import sys
sys.stdout = open('log.txt', 'a')        # Redirects prints to a file
...
print(x, y, x)                           # Shows up in log.txt
```

Here, we reset `sys.stdout` to a manually opened file named *log.txt*, located in the script's working directory and opened in append mode (so we add to its current content). After the reset, every `print` operation anywhere in the program will write its text to the end of the file *log.txt* instead of to the original output stream. The `print` operations are happy to keep calling `sys.stdout`'s `write` method, no matter what `sys.stdout` happens to refer to. Because there is just one `sys` module in your process, assigning `sys.stdout` this way will redirect every `print` anywhere in your program.

In fact, as this chapter's upcoming sidebar about `print` and `stdout` will explain, you can even reset `sys.stdout` to an object that isn't a file at all, as long as it has the expected interface: a method named `write` to receive the printed text string argument. When that object is a *class*, printed text can be routed and processed arbitrarily per a `write` method you code yourself.

This trick of resetting the output stream is primarily useful for programs originally coded with `print` statements. If you know that output should go to a file to begin with, you can always call file write methods instead. To redirect the output of a `print`-based

program, though, resetting `sys.stdout` provides a convenient alternative to changing every `print` statement or using system shell-based redirection syntax.

Automatic stream redirection

This technique of redirecting printed text by assigning `sys.stdout` is commonly used in practice. One potential problem with the last section's code, though, is that there is no direct way to restore the original output stream should you need to switch back after printing to a file. Because `sys.stdout` is just a normal file object, you can always save it and restore it if needed:#

```
C:\misc> c:\python30\python
>>> import sys
>>> temp = sys.stdout                    # Save for restoring later
>>> sys.stdout = open('log.txt', 'a')    # Redirect prints to a file
>>> print('spam')                        # Prints go to file, not here
>>> print(1, 2, 3)
>>> sys.stdout.close()                   # Flush output to disk
>>> sys.stdout = temp                    # Restore original stream

>>> print('back here')                   # Prints show up here again
back here
>>> print(open('log.txt').read())        # Result of earlier prints
spam
1 2 3
```

As you can see, though, manual saving and restoring of the original output stream like this involves quite a bit of extra work. Because this crops up fairly often, a `print` extension is available to make it unnecessary.

In 3.0, the `file` keyword allows a single `print` call to send its text to a file's `write` method, without actually resetting `sys.stdout`. Because the redirection is temporary, normal `print` calls keep printing to the original output stream. In 2.6, a `print` statement that begins with a `>>` followed by an output file object (or other compatible object) has the same effect. For example, the following again sends printed text to a file named *log.txt*:

```
log =  open('log.txt', 'a')           # 3.0
print(x, y, z, file=log)              # Print to a file-like object
print(a, b, c)                        # Print to original stdout

log =  open('log.txt', 'a')           # 2.6
print >> log, x, y, z                 # Print to a file-like object
print a, b, c                         # Print to original stdout
```

These redirected forms of `print` are handy if you need to print to *both* files and the standard output stream in the same program. If you use these forms, however, be sure

#In both 2.6 and 3.0 you may also be able to use the `__stdout__` attribute in the `sys` module, which refers to the original value `sys.stdout` had at program startup time. You still need to restore `sys.stdout` to `sys.__stdout__` to go back to this original stream value, though. See the `sys` module documentation for more details.

to give them a file object (or an object that has the same `write` method as a file object), not a file's name string. Here is the technique in action:

```
C:\misc> c:\python30\python
>>> log = open('log.txt', 'w')
>>> print(1, 2, 3, file=log)            # 2.6: print >> log, 1, 2, 3
>>> print(4, 5, 6, file=log)
>>> log.close()
>>> print(7, 8, 9)                      # 2.6: print 7, 8, 9
7 8 9
>>> print(open('log.txt').read())
1 2 3
4 5 6
```

These extended forms of `print` are also commonly used to print error messages to the standard error stream, available to your script as the preopened file object `sys.stderr`. You can either use its file `write` methods and format the output manually, or print with redirection syntax:

```
>>> import sys
>>> sys.stderr.write(('Bad!' * 8) + '\n')
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!

>>> print('Bad!' * 8, file=sys.stderr)      # 2.6: print >> sys.stderr, 'Bad' * 8
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
```

Now that you know all about print redirections, the equivalence between printing and file `write` methods should be fairly obvious. The following interaction prints both ways in 3.0, then redirects the output to an external file to verify that the same text is printed:

```
>>> X = 1; Y = 2
>>> print(X, Y)                                      # Print: the easy way
1 2
>>> import sys                                       # Print: the hard way
>>> sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
1 2
4
>>> print(X, Y, file=open('temp1', 'w'))             # Redirect text to file

>>> open('temp2', 'w').write(str(X) + ' ' + str(Y) + '\n') # Send to file manually
4
>>> print(open('temp1', 'rb').read())                # Binary mode for bytes
b'1 2\r\n'
>>> print(open('temp2', 'rb').read())
b'1 2\r\n'
```

As you can see, unless you happen to enjoy typing, print operations are usually the best option for displaying text. For another example of the equivalence between prints and file writes, watch for a 3.0 `print` function emulation example in Chapter 18; it uses this code pattern to provide a general 3.0 `print` function equivalent for use in Python 2.6.

Version-Neutral Printing

Finally, if you cannot restrict your work to Python 3.0 but still want your prints to be compatible with 3.0, you have some options. For one, you can code 2.6 `print` statements and let 3.0's `2to3` conversion script translate them to 3.0 function calls automatically. See the Python 3.0 documentation for more details about this script; it attempts to translate 2.X code to run under 3.0.

Alternatively, you can code 3.0 `print` function calls in your 2.6 code, by enabling the function call variant with a statement like the following:

```
from __future__ import print_function
```

This statement changes 2.6 to support 3.0's `print` functions exactly. This way, you can use 3.0 print features and won't have to change your prints if you later migrate to 3.0.

Also keep in mind that simple prints, like those in the first row of Table 11-5, work in *either* version of Python—because any expression may be enclosed in parentheses, we can always pretend to be calling a 3.0 `print` function in 2.6 by adding outer parentheses. The only downside to this is that it makes a tuple out of your printed objects if there are more than one—they will print with extra enclosing parentheses. In 3.0, for example, any number of objects may be listed in the call's parentheses:

```
C:\misc> c:\python30\python
>>> print('spam')                    # 3.0 print function call syntax
spam
>>> print('spam', 'ham', 'eggs')     # These are mutiple argments
spam ham eggs
```

The first of these works the same in 2.6, but the second generates a tuple in the output:

```
C:\misc> c:\python26\python
>>> print('spam')                    # 2.6 print statement, enclosing parens
spam
>>> print('spam', 'ham', 'eggs')     # This is really a tuple object!
('spam', 'ham', 'eggs')
```

To be truly portable, you can format the print string as a single object, using the string formatting expression or method call, or other string tools that we studied in Chapter 7:

```
>>> print('%s %s %s' % ('spam', 'ham', 'eggs'))
spam ham eggs
>>> print('{0} {1} {2}'.format('spam', 'ham', 'eggs'))
spam ham eggs
```

Of course, if you can use 3.0 exclusively you can forget such mappings entirely, but many Python programmers will at least encounter, if not write, 2.X code and systems for some time to come.

I use Python 3.0 `print` function calls throughout this book. I'll usually warn you that the results may have extra enclosing parentheses in 2.6 because multiple items are a tuple, but I sometimes don't, so please consider this note a blanket warning—if you see extra parentheses in your printed text in 2.6, either drop the parentheses in your `print` statements, recode your prints using the version-neutral scheme outlined here, or learn to love superfluous text.

Why You Will Care: print and stdout

The equivalence between the `print` operation and writing to `sys.stdout` is important. It makes it possible to reassign `sys.stdout` to any user-defined object that provides the same `write` method as files. Because the `print` statement just sends text to the `sys.stdout.write` method, you can capture printed text in your programs by assigning `sys.stdout` to an object whose `write` method processes the text in arbitrary ways.

For instance, you can send printed text to a GUI window, or tee it off to multiple destinations, by defining an object with a `write` method that does the required routing. You'll see an example of this trick when we study classes in Part VI of this book, but abstractly, it looks like this:

```
class FileFaker:
    def write(self, string):
        # Do something with printed text in string

import sys
sys.stdout = FileFaker()
print(someObjects)              # Sends to class write method
```

This works because `print` is what we will call in the next part of this book a *polymorphic* operation—it doesn't care what `sys.stdout` is, only that it has a method (i.e., interface) called `write`. This redirection to objects is made even simpler with the `file` keyword argument in 3.0 and the `>>` extended form of `print` in 2.6, because we don't need to reset `sys.stdout` explicitly—normal prints will still be routed to the `stdout` stream:

```
myobj = FileFaker()             # 3.0: Redirect to object for one print
print(someObjects, file=myobj)  # Does not reset sys.stdout

myobj = FileFaker()             # 2.6: same effect
print >> myobj, someObjects     # Does not reset sys.stdout
```

Python's built-in `input` function reads from the `sys.stdin` file, so you can intercept read requests in a similar way, using classes that implement file-like `read` methods instead. See the `input` and `while` loop example in Chapter 10 for more background on this.

Notice that because printed text goes to the `stdout` stream, it's the way to print HTML in CGI scripts used on the Web. It also enables you to redirect Python script input and output at the operating system's shell command line, as usual:

```
python script.py < inputfile > outputfile
python script.py | filterProgram
```

Python's print operation redirection tools are essentially pure-Python alternatives to these shell syntax forms.

Chapter Summary

In this chapter, we began our in-depth look at Python statements by exploring assignments, expressions, and print operations. Although these are generally simple to use, they have some alternative forms that, while optional, are often convenient in practice: augmented assignment statements and the redirection form of `print` operations, for example, allow us to avoid some manual coding work. Along the way, we also studied the syntax of variable names, stream redirection techniques, and a variety of common mistakes to avoid, such as assigning the result of an `append` method call back to a variable.

In the next chapter, we'll continue our statement tour by filling in details about the `if` statement, Python's main selection tool; there, we'll also revisit Python's syntax model in more depth and look at the behavior of Boolean expressions. Before we move on, though, the end-of-chapter quiz will test your knowledge of what you've learned here.

Test Your Knowledge: Quiz

1. Name three ways that you can assign three variables to the same value.
2. Why might you need to care when assigning three variables to a mutable object?
3. What's wrong with saying `L = L.sort()`?
4. How might you use the `print` operation to send text to an external file?

Test Your Knowledge: Answers

1. You can use multiple-target assignments (`A = B = C = 0`), sequence assignment (`A, B, C = 0, 0, 0`), or multiple assignment statements on three separate lines (`A = 0`, `B = 0`, and `C = 0`). With the latter technique, as introduced in Chapter 10, you can also string the three separate statements together on the same line by separating them with semicolons (`A = 0; B = 0; C = 0`).

2. If you assign them this way:

   ```
   A = B = C = []
   ```

   all three names reference the same object, so changing it in-place from one (e.g., `A.append(99)`) will affect the others. This is true only for in-place changes to mutable objects like lists and dictionaries; for immutable objects such as numbers and strings, this issue is irrelevant.
3. The list `sort` method is like `append` in that it makes an in-place change to the subject list—it returns `None`, not the list it changes. The assignment back to `L` sets `L` to `None`, not to the sorted list. As we'll see later in this part of the book, a newer built-in function, `sorted`, sorts any sequence and returns a new list with the sorting result; because this is not an in-place change, its result can be meaningfully assigned to a name.
4. To print to a file for a single `print` operation, you can use 3.0's `print(X, file=F)` call form, use 2.6's extended `print >> file, X` statement form, or assign `sys.stdout` to a manually opened file before the `print` and restore the original after. You can also redirect all of a program's printed text to a file with special syntax in the system shell, but this is outside Python's scope.

CHAPTER 12

if Tests and Syntax Rules

This chapter presents the Python `if` statement, which is the main statement used for selecting from alternative actions based on test results. Because this is our first in-depth look at *compound statements*—statements that embed other statements—we will also explore the general concepts behind the Python statement syntax model here in more detail than we did in the introduction in Chapter 10. Because the `if` statement introduces the notion of tests, this chapter will also deal with Boolean expressions and fill in some details on truth tests in general.

if Statements

In simple terms, the Python `if` statement selects actions to perform. It's the primary selection tool in Python and represents much of the *logic* a Python program possesses. It's also our first compound statement. Like all compound Python statements, the `if` statement may contain other statements, including other `if`s. In fact, Python lets you combine statements in a program sequentially (so that they execute one after another), and in an arbitrarily nested fashion (so that they execute only under certain conditions).

General Format

The Python `if` statement is typical of `if` statements in most procedural languages. It takes the form of an `if` test, followed by one or more optional `elif` ("else if") tests and a final optional `else` block. The tests and the `else` part each have an associated block of nested statements, indented under a header line. When the `if` statement runs, Python executes the block of code associated with the first test that evaluates to true, or the `else` block if all tests prove false. The general form of an `if` statement looks like this:

```
if <test1>:              # if test
    <statements1>        # Associated block
elif <test2>:            # Optional elifs
    <statements2>
else:                    # Optional else
    <statements3>
```

Basic Examples

To demonstrate, let's look at a few simple examples of the `if` statement at work. All parts are optional, except the initial `if` test and its associated statements. Thus, in the simplest case, the other parts are omitted:

```
>>> if 1:
...     print('true')
...
true
```

Notice how the prompt changes to `...` for continuation lines when typing interactively in the basic interface used here; in IDLE, you'll simply drop down to an indented line instead (hit Backspace to back up). A blank line (which you can get by pressing Enter twice) terminates and runs the entire statement. Remember that `1` is Boolean true, so this statement's test always succeeds. To handle a false result, code the `else`:

```
>>> if not 1:
...     print('true')
... else:
...     print('false')
...
false
```

Multiway Branching

Now here's an example of a more complex `if` statement, with all its optional parts present:

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...     print("how's jessica?")
... elif x == 'bugs':
...     print("what's up doc?")
... else:
...     print('Run away! Run away!')
...
Run away! Run away!
```

This multiline statement extends from the `if` line through the `else` block. When it's run, Python executes the statements nested under the first test that is true, or the `else` part if all tests are false (in this example, they are). In practice, both the `elif` and `else` parts may be omitted, and there may be more than one statement nested in each section. Note that the words `if`, `elif`, and `else` are associated by the fact that they line up vertically, with the same indentation.

If you've used languages like C or Pascal, you might be interested to know that there is no `switch` or `case` statement in Python that selects an action based on a variable's value. Instead, *multiway branching* is coded either as a series of `if`/`elif` tests, as in the prior example, or by indexing dictionaries or searching lists. Because dictionaries and lists can be built at runtime, they're sometimes more flexible than hardcoded `if` logic:

```
>>> choice = 'ham'
>>> print({'spam':  1.25,          # A dictionary-based 'switch'
...        'ham':   1.99,          # Use has_key or get for default
...        'eggs':  0.99,
...        'bacon': 1.10}[choice])
1.99
```

Although it may take a few moments for this to sink in the first time you see it, this dictionary is a multiway branch—indexing on the key `choice` branches to one of a set of values, much like a `switch` in C. An almost equivalent but more verbose Python `if` statement might look like this:

```
>>> if choice == 'spam':
...     print(1.25)
... elif choice == 'ham':
...     print(1.99)
... elif choice == 'eggs':
...     print(0.99)
... elif choice == 'bacon':
...     print(1.10)
... else:
...     print('Bad choice')
...
1.99
```

Notice the `else` clause on the `if` here to handle the default case when no key matches. As we saw in Chapter 8, dictionary defaults can be coded with `in` expressions, `get` method calls, or exception catching. All of the same techniques can be used here to code a default action in a dictionary-based multiway branch. Here's the `get` scheme at work with defaults:

```
>>> branch = {'spam': 1.25,
...           'ham':  1.99,
...           'eggs': 0.99}

>>> print(branch.get('spam', 'Bad choice'))
1.25
>>> print(branch.get('bacon', 'Bad choice'))
Bad choice
```

An `in` membership test in an `if` statement can have the same default effect:

```
>>> choice = 'bacon'
>>> if choice in branch:
...     print(branch[choice])
... else:
...     print('Bad choice')
...
Bad choice
```

Dictionaries are good for associating values with keys, but what about the more complicated actions you can code in the statement blocks associated with `if` statements? In Part IV, you'll learn that dictionaries can also contain *functions* to represent more complex branch actions and implement general jump tables. Such functions appear as

dictionary values, may be coded as function names or `lambda`s, and are called by adding parentheses to trigger their actions; stay tuned for more on this topic in Chapter 19.

Although dictionary-based multiway branching is useful in programs that deal with more dynamic data, most programmers will probably find that coding an `if` statement is the most straightforward way to perform multiway branching. As a rule of thumb in coding, when in doubt, err on the side of simplicity and readability; it's the "Pythonic" way.

Python Syntax Rules

I introduced Python's syntax model in Chapter 10. Now that we're stepping up to larger statements like the `if`, this section reviews and expands on the syntax ideas introduced earlier. In general, Python has a simple, statement-based syntax. However, there are a few properties you need to know about:

- **Statements execute one after another, until you say otherwise**. Python normally runs statements in a file or nested block in order from first to last, but statements like `if` (and, as you'll see, loops) cause the interpreter to jump around in your code. Because Python's path through a program is called the *control flow*, statements such as `if` that affect it are often called *control-flow statements*.
- **Block and statement boundaries are detected automatically**. As we've seen, there are no braces or "begin/end" delimiters around blocks of code in Python; instead, Python uses the indentation of statements under a header to group the statements in a nested block. Similarly, Python statements are not normally terminated with semicolons; rather, the end of a line usually marks the end of the statement coded on that line.
- **Compound statements = header + ":" + indented statements**. All compound statements in Python follow the same pattern: a header line terminated with a colon, followed by one or more nested statements, usually indented under the header. The indented statements are called a *block* (or sometimes, a suite). In the `if` statement, the `elif` and `else` clauses are part of the `if`, but they are also header lines with nested blocks of their own.
- **Blank lines, spaces, and comments are usually ignored**. Blank lines are ignored in files (but not at the interactive prompt, when they terminate compound statements). Spaces inside statements and expressions are almost always ignored (except in string literals, and when used for indentation). Comments are always ignored: they start with a `#` character (not inside a string literal) and extend to the end of the current line.
- **Docstrings are ignored but are saved and displayed by tools**. Python supports an additional comment form called documentation strings (*docstrings* for short), which, unlike `#` comments, are retained at runtime for inspection. Docstrings are simply strings that show up at the top of program files and some statements. Python

ignores their contents, but they are automatically attached to objects at runtime and may be displayed with documentation tools. Docstrings are part of Python's larger documentation strategy and are covered in the last chapter in this part of the book.

As you've seen, there are no variable type declarations in Python; this fact alone makes for a much simpler language syntax than what you may be used to. However, for most new users the lack of the braces and semicolons used to mark blocks and statements in many other languages seems to be the most novel syntactic feature of Python, so let's explore what this means in more detail.

Block Delimiters: Indentation Rules

Python detects block boundaries automatically, by line *indentation*—that is, the empty space to the left of your code. All statements indented the same distance to the right belong to the same block of code. In other words, the statements within a block line up vertically, as in a column. The block ends when the end of the file or a lesser-indented line is encountered, and more deeply nested blocks are simply indented further to the right than the statements in the enclosing block.

For instance, Figure 12-1 demonstrates the block structure of the following code:

```
x = 1
if x:
    y = 2
    if y:
        print('block2')
    print('block1')
print('block0')
```

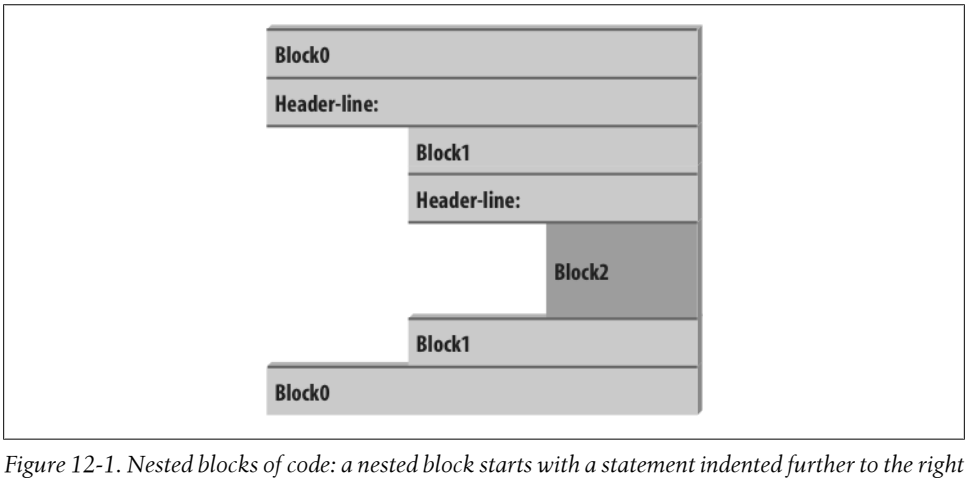

Figure 12-1. Nested blocks of code: a nested block starts with a statement indented further to the right and ends with either a statement that is indented less, or the end of the file.

This code contains three blocks: the first (the top-level code of the file) is not indented at all, the second (within the outer `if` statement) is indented four spaces, and the third (the `print` statement under the nested `if`) is indented eight spaces.

In general, top-level (unnested) code must start in column 1. Nested blocks can start in any column; indentation may consist of any number of spaces and tabs, as long as it's the same for all the statements in a given single block. That is, Python doesn't care *how* you indent your code; it only cares that it's done consistently. Four spaces or one tab per indentation level are common conventions, but there is no absolute standard in the Python world.

Indenting code is quite natural in practice. For example, the following (arguably silly) code snippet demonstrates common indentation errors in Python code:

```
  x = 'SPAM'                      # Error: first line indented
if 'rubbery' in 'shrubbery':
    print(x * 8)
        x += 'NI'                 # Error: unexpected indentation
        if x.endswith('NI'):
                x *= 2
            print(x)              # Error: inconsistent indentation
```

The properly indented version of this code looks like the following—even for an artificial example like this, proper indentation makes the code's intent much more apparent:

```
x = 'SPAM'
if 'rubbery' in 'shrubbery':
    print(x * 8)
    x += 'NI'
    if x.endswith('NI'):
        x *= 2
        print(x)                  # Prints "SPAMNISPAMNI"
```

It's important to know that the only major place in Python where whitespace matters is where it's used to the left of your code, for indentation; in most other contexts, space can be coded or not. However, indentation is really part of Python syntax, not just a stylistic suggestion: all the statements within any given single block must be indented to the same level, or Python reports a syntax error. This is intentional—because you don't need to explicitly mark the start and end of a nested block of code, some of the syntactic clutter found in other languages is unnecessary in Python.

As described in Chapter 10, making indentation part of the syntax model also enforces consistency, a crucial component of readability in structured programming languages like Python. Python's syntax is sometimes described as "what you see is what you get"—the indentation of each line of code unambiguously tells readers what it is associated with. This uniform and consistent appearance makes Python code easier to maintain and reuse.

Indentation is more natural than the details might imply, and it makes your code reflect its logical structure. Consistently indented code always satisfies Python's rules. Moreover, most text editors (including IDLE) make it easy to follow Python's indentation model by automatically indenting code as you type it.

Avoid mixing tabs and spaces: New error checking in 3.0

One rule of thumb: although you can use spaces or tabs to indent, it's usually not a good idea to mix the two within a block—use one or the other. Technically, tabs count for enough spaces to move the current column number up to a multiple of 8, and your code will work if you mix tabs and spaces consistently. However, such code can be difficult to change. Worse, mixing tabs and spaces makes your code difficult to read—tabs may look very different in the next programmer's editor than they do in yours.

In fact, Python 3.0 now issues an error, for these very reasons, when a script mixes tabs and spaces for indentation inconsistently within a block (that is, in a way that makes it dependent on a tab's equivalent in spaces). Python 2.6 allows such scripts to run, but it has a `-t` command-line flag that will warn you about inconsistent tab usage and a `-tt` flag that will issue errors for such code (you can use these switches in a command line like `python -t main.py` in a system shell window). Python 3.0's error case is equivalent to 2.6's `-tt` switch.

Statement Delimiters: Lines and Continuations

A statement in Python normally ends at the end of the line on which it appears. When a statement is too long to fit on a single line, though, a few special rules may be used to make it span multiple lines:

- **Statements may span multiple lines if you're continuing an open syntactic pair**. Python lets you continue typing a statement on the next line if you're coding something enclosed in a `()`, `{}`, or `[]` pair. For instance, expressions in parentheses and dictionary and list literals can span any number of lines; your statement doesn't end until the Python interpreter reaches the line on which you type the closing part of the pair (a `)`, `}`, or `]`). Continuation lines (lines 2 and beyond of the statement) can start at any indentation level you like, but you should try to make them align vertically for readability if possible. This open pairs rule also covers set and dictionary comprehensions in Python 3.0.
- **Statements may span multiple lines if they end in a backslash**. This is a somewhat outdated feature, but if a statement needs to span multiple lines, you can also add a backslash (a `\` not embedded in a string literal or comment) at the end of the prior line to indicate you're continuing on the next line. Because you can also continue by adding parentheses around most constructs, backslashes are almost never used. This approach is error-prone: accidentally forgetting a `\` usually generates a syntax error and might even cause the next line to be silently mistaken to be a new statement, with unexpected results.

- **Special rules for string literals**. As we learned in Chapter 7, triple-quoted string blocks are designed to span multiple lines normally. We also learned in Chapter 7 that adjacent string literals are implicitly concatenated; when used in conjunction with the open pairs rule mentioned earlier, wrapping this construct in parentheses allows it to span multiple lines.
- **Other rules**. There are a few other points to mention with regard to statement delimiters. Although uncommon, you can terminate a statement with a semicolon—this convention is sometimes used to squeeze more than one simple (noncompound) statement onto a single line. Also, comments and blank lines can appear anywhere in a file; comments (which begin with a `#` character) terminate at the end of the line on which they appear.

A Few Special Cases

Here's what a continuation line looks like using the open syntactic pairs rule. Delimited constructs, such as lists in square brackets, can span across any number of lines:

```
L = ["Good",
     "Bad",
     "Ugly"]                    # Open pairs may span lines
```

This also works for anything in parentheses (expressions, function arguments, function headers, tuples, and generator expressions), as well as anything in curly braces (dictionaries and, in 3.0, set literals and set and dictionary comprehensions). Some of these are tools we'll study in later chapters, but this rule naturally covers most constructs that span lines in practice.

If you like using backslashes to continue lines, you can, but it's not common practice in Python:

```
if a == b and c == d and   \
   d == e and f == g:
   print('olde')                # Backslashes allow continuations...
```

Because any expression can be enclosed in parentheses, you can usually use the open pairs technique instead if you need your code to span multiple lines—simply wrap a part of your statement in parentheses:

```
if (a == b and c == d and
    d == e and e == f):
    print('new')                # But parentheses usually do too
```

In fact, backslashes are frowned on, because they're too easy to not notice and too easy to omit altogether. In the following, `x` is assigned `10` with the backslash, as intended; if the backslash is accidentally omitted, though, `x` is assigned `6` instead, and no error is reported (the `+4` is a valid expression statement by itself).

In a real program with a more complex assignment, this could be the source of a very nasty bug:*

```
x = 1 + 2 + 3 \                    # Omitting the \ makes this very different
+4
```

As another special case, Python allows you to write more than one noncompound statement (i.e., statements without nested statements) on the same line, separated by semicolons. Some coders use this form to save program file real estate, but it usually makes for more readable code if you stick to one statement per line for most of your work:

```
x = 1; y = 2; print(x)             # More than one simple statement
```

As we learned in Chapter 7, triple-quoted string literals span lines too. In addition, if two string literals appear next to each other, they are concatenated as if a `+` had been added between them—when used in conjunction with the open pairs rule, wrapping in parentheses allows this form to span multiple lines. For example, the first of the following inserts newline characters at line breaks and assigns `S` to `'\naaaa\nbbbb\ncccc'`, and the second implicitly concatenates and assigns `S` to `'aaaabbbbcccc'`; comments are ignored in the second form, but included in the string in the first:

```
S = """
aaaa
bbbb
cccc"""

S = ('aaaa'
     'bbbb'                        # Comments here are ignored
     'cccc')
```

Finally, Python lets you move a compound statement's body up to the header line, provided the body is just a simple (noncompound) statement. You'll most often see this used for simple `if` statements with a single test and action:

```
if 1: print('hello')               # Simple statement on header line
```

You can combine some of these special cases to write code that is difficult to read, but I don't recommend it; as a rule of thumb, try to keep each statement on a line of its own, and indent all but the simplest of blocks. Six months down the road, you'll be happy you did.

* Frankly, it's surprising that this wasn't removed in Python 3.0, given some of its other changes! (See Table P-2 of the Preface for a list of 3.0 removals; some seem fairly innocuous in comparison with the dangers inherent in backslash continuations.) Then again, this book's goal is Python instruction, not populist outrage, so the best advice I can give is simply: don't do this.

Truth Tests

The notions of comparison, equality, and truth values were introduced in Chapter 9. Because the `if` statement is the first statement we've looked at that actually uses test results, we'll expand on some of these ideas here. In particular, Python's Boolean operators are a bit different from their counterparts in languages like C. In Python:

- Any nonzero number or nonempty object is true.
- Zero numbers, empty objects, and the special object `None` are considered false.
- Comparisons and equality tests are applied recursively to data structures.
- Comparisons and equality tests return `True` or `False` (custom versions of `1` and `0`).
- Boolean `and` and `or` operators return a true or false operand object.

In short, Boolean operators are used to combine the results of other tests. There are three Boolean expression operators in Python:

`X and Y`
: Is true if both `X` and `Y` are true

`X or Y`
: Is true if either `X` or `Y` is true

`not X`
: Is true if `X` is false (the expression returns `True` or `False`)

Here, `X` and `Y` may be any truth value, or any expression that returns a truth value (e.g., an equality test, range comparison, and so on). Boolean operators are typed out as words in Python (instead of C's `&&`, `||`, and `!`). Also, Boolean `and` and `or` operators return a true or false *object* in Python, not the values `True` or `False`. Let's look at a few examples to see how this works:

```
>>> 2 < 3, 3 < 2          # Less-than: return True or False (1 or 0)
(True, False)
```

Magnitude comparisons such as these return `True` or `False` as their truth results, which, as we learned in Chapters 5 and 9, are really just custom versions of the integers `1` and `0` (they print themselves differently but are otherwise the same).

On the other hand, the `and` and `or` operators always return an object—either the object on the left side of the operator or the object on the right. If we test their results in `if` or other statements, they will be as expected (remember, every object is inherently true or false), but we won't get back a simple `True` or `False`.

For `or` tests, Python evaluates the operand objects from left to right and returns the first one that is true. Moreover, Python stops at the first true operand it finds. This is usually called *short-circuit evaluation*, as determining a result short-circuits (terminates) the rest of the expression:

```
>>> 2 or 3, 3 or 2      # Return left operand if true
(2, 3)                  # Else, return right operand (true or false)
>>> [] or 3
3
>>> [] or {}
{}
```

In the first line of the preceding example, both operands (`2` and `3`) are true (i.e., are nonzero), so Python always stops and returns the one on the left. In the other two tests, the left operand is false (an empty object), so Python simply evaluates and returns the object on the right (which may happen to have either a true or a false value when tested).

`and` operations also stop as soon as the result is known; however, in this case Python evaluates the operands from left to right and stops at the first *false* object:

```
>>> 2 and 3, 3 and 2    # Return left operand if false
(3, 2)                  # Else, return right operand (true or false)
>>> [] and {}
[]
>>> 3 and []
[]
```

Here, both operands are true in the first line, so Python evaluates both sides and returns the object on the right. In the second test, the left operand is false (`[]`), so Python stops and returns it as the test result. In the last test, the left side is true (`3`), so Python evaluates and returns the object on the right (which happens to be a false `[]`).

The end result of all this is the same as in C and most other languages—you get a value that is logically true or false if tested in an `if` or `while`. However, in Python Booleans return either the left or the right object, not a simple integer flag.

This behavior of `and` and `or` may seem esoteric at first glance, but see this chapter's sidebar "Why You Will Care: Booleans" on page 323 for examples of how it is sometimes used to advantage in coding by Python programmers. The next section also shows a common way to leverage this behavior, and its replacement in more recent versions of Python.

The if/else Ternary Expression

One common role for the prior section's Boolean operators is to code an expression that runs the same as an `if` statement. Consider the following statement, which sets `A` to either `Y` or `Z`, based on the truth value of `X`:

```
if X:
    A = Y
else:
    A = Z
```

Sometimes, though, the items involved in such a statement are so simple that it seems like overkill to spread them across four lines. At other times, we may want to nest such a construct in a larger statement instead of assigning its result to a variable. For these reasons (and, frankly, because the C language has a similar tool†), Python 2.5 introduced a new expression format that allows us to say the same thing in one expression:

```
A = Y if X else Z
```

This expression has the exact same effect as the preceding four-line `if` statement, but it's simpler to code. As in the statement equivalent, Python runs expression `Y` only if `X` turns out to be true, and runs expression `Z` only if `X` turns out to be false. That is, it *short-circuits*, just like the Boolean operators described in the prior section. Here are some examples of it in action:

```
>>> A = 't' if 'spam' else 'f'       # Nonempty is true
>>> A
't'
>>> A = 't' if '' else 'f'
>>> A
'f'
```

Prior to Python 2.5 (and after 2.5, if you insist), the same effect can often be achieved by a careful combination of the `and` and `or` operators, because they return either the object on the left side or the object on the right:

```
A = ((X and Y) or Z)
```

This works, but there is a catch—you have to be able to assume that `Y` will be Boolean true. If that is the case, the effect is the same: the `and` runs first and returns `Y` if `X` is true; if it's not, the `or` simply returns `Z`. In other words, we get "if `X` then `Y` else `Z`."

This `and`/`or` combination also seems to require a "moment of great clarity" to understand the first time you see it, and it's no longer required as of 2.5—use the equivalent and more robust and mnemonic `Y if X else Z` instead if you need this as an expression, or use a full `if` statement if the parts are nontrivial.

As a side note, using the following expression in Python is similar because the `bool` function will translate `X` into the equivalent of integer `1` or `0`, which can then be used to pick true and false values from a list:

```
A = [Z, Y][bool(X)]
```

† In fact, Python's `X if Y else Z` has a slightly different order than C's `Y ? X : Z`. This was reportedly done in response to analysis of common use patterns in Python code. According to rumor, this order was also chosen in part to discourage ex-C programmers from overusing it! Remember, simple is better than complex, in Python and elsewhere.

For example:

```
>>> ['f', 't'][bool('')]
'f'
>>> ['f', 't'][bool('spam')]
't'
```

However, this isn't exactly the same, because Python will not short-circuit—it will always run both `Z` and `Y`, regardless of the value of `X`. Because of such complexities, you're better off using the simpler and more easily understood `if`/`else` expression as of Python 2.5 and later. Again, though, you should use even that sparingly, and only if its parts are all fairly simple; otherwise, you're better off coding the full `if` statement form to make changes easier in the future. Your coworkers will be happy you did.

Still, you may see the `and`/`or` version in code written prior to 2.5 (and in code written by C programmers who haven't quite let go of their dark coding pasts...).

Why You Will Care: Booleans

One common way to use the somewhat unusual behavior of Python Boolean operators is to select from a set of objects with an `or`. A statement such as this:

```
X = A or B or C or None
```

sets `X` to the first nonempty (that is, true) object among `A`, `B`, and `C`, or to `None` if all of them are empty. This works because the `or` operator returns one of its two objects, and it turns out to be a fairly common coding paradigm in Python: to select a nonempty object from among a fixed-size set, simply string them together in an `or` expression. In simpler form, this is also commonly used to designate a default—the following sets `X` to `A` if `A` is true (or nonempty), and to `default` otherwise:

```
X = A or default
```

It's also important to understand short-circuit evaluation because expressions on the right of a Boolean operator might call functions that perform substantial or important work, or have side effects that won't happen if the short-circuit rule takes effect:

```
if f1() or f2(): ...
```

Here, if `f1` returns a true (or nonempty) value, Python will never run `f2`. To guarantee that both functions will be run, call them before the `or`:

```
tmp1, tmp2 = f1(), f2()
if tmp1 or tmp2: ...
```

You've already seen another application of this behavior in this chapter: because of the way Booleans work, the expression `((A and B) or C)` can be used to emulate an `if`/`else` statement—almost (see this chapter's discussion of this form for details).

We met additional Boolean use cases in prior chapters. As we saw in Chapter 9, because all objects are inherently true or false, it's common and easier in Python to test an object directly (`if X:`) than to compare it to an empty value (`if X != '':`). For a string, the two tests are equivalent. As we also saw in Chapter 5, the preset Booleans values `True` and `False` are the same as the integers `1` and `0` and are useful for initializing variables

(`X = False`), for loop tests (`while True:`), and for displaying results at the interactive prompt.

Also watch for the discussion of operator overloading in Part VI: when we define new object types with classes, we can specify their Boolean nature with either the `__bool__` or `__len__` methods (`__bool__` is named `__nonzero__` in 2.6). The latter of these is tried if the former is absent and designates false by returning a length of zero—an empty object is considered false.

Chapter Summary

In this chapter, we studied the Python `if` statement. Additionally, because this was our first compound and logical statement, we reviewed Python's general syntax rules and explored the operation of truth tests in more depth than we were able to previously. Along the way, we also looked at how to code multiway branching in Python and learned about the `if`/`else` expression introduced in Python 2.5.

The next chapter continues our look at procedural statements by expanding on the `while` and `for` loops. There, we'll learn about alternative ways to code loops in Python, some of which may be better than others. Before that, though, here is the usual chapter quiz.

Test Your Knowledge: Quiz

1. How might you code a multiway branch in Python?
2. How can you code an `if`/`else` statement as an expression in Python?
3. How can you make a single statement span many lines?
4. What do the words `True` and `False` mean?

Test Your Knowledge: Answers

1. An `if` statement with multiple `elif` clauses is often the most straightforward way to code a multiway branch, though not necessarily the most concise. Dictionary indexing can often achieve the same result, especially if the dictionary contains callable functions coded with `def` statements or `lambda` expressions.
2. In Python 2.5 and later, the expression form `Y if X else Z` returns `Y` if `X` is true, or `Z` otherwise; it's the same as a four-line `if` statement. The `and`/`or` combination (`((X and Y) or Z)`) can work the same way, but it's more obscure and requires that the `Y` part be true.

3. Wrap up the statement in an open syntactic pair (`()`, `[]`, or `{}`), and it can span as many lines as you like; the statement ends when Python sees the closing (right) half of the pair, and lines 2 and beyond of the statement can begin at any indentation level.
4. `True` and `False` are just custom versions of the integers `1` and `0`, respectively: they always stand for Boolean true and false values in Python. They're available for use in truth tests and variable initialization and are printed for expression results at the interactive prompt.

CHAPTER 13
while and for Loops

This chapter concludes our tour of Python procedural statements by presenting the language's two main *looping* constructs—statements that repeat an action over and over. The first of these, the `while` statement, provides a way to code general loops. The second, the `for` statement, is designed for stepping through the items in a sequence object and running a block of code for each.

We've seen both of these informally already, but we'll fill in additional usage details here. While we're at it, we'll also study a few less prominent statements used within loops, such as `break` and `continue`, and cover some built-ins commonly used with loops, such as `range`, `zip`, and `map`.

Although the `while` and `for` statements covered here are the primary syntax provided for coding repeated actions, there are additional looping operations and concepts in Python. Because of that, the iteration story is continued in the next chapter, where we'll explore the related ideas of Python's *iteration protocol* (used by the `for` loop) and *list comprehensions* (a close cousin to the `for` loop). Later chapters explore even more exotic iteration tools such as *generators*, `filter`, and `reduce`. For now, though, let's keep things simple.

while Loops

Python's `while` statement is the most general iteration construct in the language. In simple terms, it repeatedly executes a block of (normally indented) statements as long as a test at the top keeps evaluating to a true value. It is called a "loop" because control keeps looping back to the start of the statement until the test becomes false. When the test becomes false, control passes to the statement that follows the `while` block. The net effect is that the loop's body is executed repeatedly while the test at the top is true; if the test is false to begin with, the body never runs.

General Format

In its most complex form, the `while` statement consists of a header line with a test expression, a body of one or more indented statements, and an optional `else` part that is executed if control exits the loop without a `break` statement being encountered. Python keeps evaluating the test at the top and executing the statements nested in the loop body until the test returns a false value:

```
while <test>:                  # Loop test
    <statements1>              # Loop body
else:                          # Optional else
    <statements2>              # Run if didn't exit loop with break
```

Examples

To illustrate, let's look at a few simple `while` loops in action. The first, which consists of a `print` statement nested in a `while` loop, just prints a message forever. Recall that `True` is just a custom version of the integer 1 and always stands for a Boolean true value; because the test is always true, Python keeps executing the body forever, or until you stop its execution. This sort of behavior is usually called an *infinite loop*:

```
>>> while True:
...     print('Type Ctrl-C to stop me!')
```

The next example keeps slicing off the first character of a string until the string is empty and hence false. It's typical to test an object directly like this instead of using the more verbose equivalent (`while x != '':`). Later in this chapter, we'll see other ways to step more directly through the items in a string with a `for` loop.

```
>>> x = 'spam'
>>> while x:                   # While x is not empty
...     print(x, end=' ')
...     x = x[1:]              # Strip first character off x
...
spam pam am m
```

Note the `end=' '` keyword argument used here to place all outputs on the same line separated by a space; see Chapter 11 if you've forgotten why this works as it does. The following code counts from the value of `a` up to, but not including, `b`. We'll see an easier way to do this with a Python `for` loop and the built-in `range` function later:

```
>>> a=0; b=10
>>> while a < b:               # One way to code counter loops
...     print(a, end=' ')
...     a += 1                 # Or, a = a + 1
...
0 1 2 3 4 5 6 7 8 9
```

Finally, notice that Python doesn't have what some languages call a "do until" loop statement. However, we can simulate one with a test and `break` at the bottom of the loop body:

```
while True:
    ...loop body...
    if exitTest(): break
```

To fully understand how this structure works, we need to move on to the next section and learn more about the `break` statement.

break, continue, pass, and the Loop else

Now that we've seen a few Python loops in action, it's time to take a look at two simple statements that have a purpose only when nested inside loops—the `break` and continue statements. While we're looking at oddballs, we will also study the loop `else` clause here, because it is intertwined with `break`, and Python's empty placeholder statement, the `pass` (which is not tied to loops per se, but falls into the general category of simple one-word statements). In Python:

`break`
: Jumps out of the closest enclosing loop (past the entire loop statement)

`continue`
: Jumps to the top of the closest enclosing loop (to the loop's header line)

`pass`
: Does nothing at all: it's an empty statement placeholder

`Loop else block`
: Runs if and only if the loop is exited normally (i.e., without hitting a `break`)

General Loop Format

Factoring in `break` and `continue` statements, the general format of the `while` loop looks like this:

```
while <test1>:
    <statements1>
    if <test2>: break              # Exit loop now, skip else
    if <test3>: continue           # Go to top of loop now, to test1
else:
    <statements2>                  # Run if we didn't hit a 'break'
```

`break` and `continue` statements can appear anywhere inside the `while` (or `for`) loop's body, but they are usually coded further nested in an `if` test to take action in response to some condition.

Let's turn to a few simple examples to see how these statements come together in practice.

pass

Simple things first: the `pass` statement is a no-operation placeholder that is used when the syntax requires a statement, but you have nothing useful to say. It is often used to code an empty body for a compound statement. For instance, if you want to code an infinite loop that does nothing each time through, do it with a `pass`:

```
while True: pass                    # Type Ctrl-C to stop me!
```

Because the body is just an empty statement, Python gets stuck in this loop. `pass` is roughly to statements as `None` is to objects—an explicit nothing. Notice that here the `while` loop's body is on the same line as the header, after the colon; as with `if` statements, this only works if the body isn't a compound statement.

This example does nothing forever. It probably isn't the most useful Python program ever written (unless you want to warm up your laptop computer on a cold winter's day!); frankly, though, I couldn't think of a better `pass` example at this point in the book.

We'll see other places where `pass` makes more sense later—for instance, to ignore exceptions caught by `try` statements, and to define empty `class` objects with attributes that behave like "structs" and "records" in other languages. A `pass` is also sometime coded to mean "to be filled in later," to stub out the bodies of functions temporarily:

```
def func1():
    pass                            # Add real code here later

def func2():
    pass
```

We can't leave the body empty without getting a syntax error, so we say `pass` instead.

Version skew note: Python 3.0 (but not 2.6) allows *ellipses* coded as `...` (literally, three consecutive dots) to appear any place an expression can. Because ellipses do nothing by themselves, this can serve as an alternative to the `pass` statement, especially for code to be filled in later—a sort of Python "TBD":

```
def func1():
    ...                             # Alternative to pass

def func2():
    ...

func1()                             # Does nothing if called
```

Ellipses can also appear on the same line as a statement header and may be used to initialize variable names if no specific type is required:

```
def func1(): ...                    # Works on same line too
def func2(): ...

>>> X = ...                         # Alternative to None
```

```
>>> X
Ellipsis
```

This notation is new in Python 3.0 (and goes well beyond the original intent of `...` in slicing extensions), so time will tell if it becomes widespread enough to challenge `pass` and `None` in these roles.

continue

The `continue` statement causes an immediate jump to the top of a loop. It also sometimes lets you avoid statement nesting. The next example uses `continue` to skip odd numbers. This code prints all even numbers less than 10 and greater than or equal to 0. Remember, 0 means false and `%` is the remainder of division operator, so this loop counts down to 0, skipping numbers that aren't multiples of 2 (it prints `8 6 4 2 0`):

```
x = 10
while x:
    x = x-1                         # Or, x -= 1
    if x % 2 != 0: continue         # Odd? -- skip print
    print(x, end=' ')
```

Because `continue` jumps to the top of the loop, you don't need to nest the `print` statement inside an `if` test; the `print` is only reached if the `continue` is not run. If this sounds similar to a "goto" in other languages, it should. Python has no "goto" statement, but because `continue` lets you jump about in a program, many of the warnings about readability and maintainability you may have heard about goto apply. `continue` should probably be used sparingly, especially when you're first getting started with Python. For instance, the last example might be clearer if the `print` were nested under the `if`:

```
x = 10
while x:
    x = x-1
    if x % 2 == 0:                  # Even? -- print
        print(x, end=' ')
```

break

The `break` statement causes an immediate exit from a loop. Because the code that follows it in the loop is not executed if the `break` is reached, you can also sometimes avoid nesting by including a `break`. For example, here is a simple interactive loop (a variant of a larger example we studied in Chapter 10) that inputs data with `input` (known as `raw_input` in Python 2.6) and exits when the user enters "stop" for the name request:

```
>>> while True:
...     name = input('Enter name:')
...     if name == 'stop': break
...     age  = input('Enter age: ')
...     print('Hello', name, '=>', int(age) ** 2)
...
Enter name:mel
Enter age: 40
```

```
Hello mel => 1600
Enter name:bob
Enter age: 30
Hello bob => 900
Enter name:stop
```

Notice how this code converts the `age` input to an integer with `int` before raising it to the second power; as you'll recall, this is necessary because `input` returns user input as a string. In Chapter 35, you'll see that `input` also raises an exception at end-of-file (e.g., if the user types Ctrl-Z or Ctrl-D); if this matters, wrap `input` in `try` statements.

Loop else

When combined with the loop `else` clause, the `break` statement can often eliminate the need for the search status flags used in other languages. For instance, the following piece of code determines whether a positive integer `y` is prime by searching for factors greater than 1:

```
x = y // 2                          # For some y > 1
while x > 1:
    if y % x == 0:                  # Remainder
        print(y, 'has factor', x)
        break                       # Skip else
    x -= 1
else:                               # Normal exit
    print(y, 'is prime')
```

Rather than setting a flag to be tested when the loop is exited, it inserts a `break` where a factor is found. This way, the loop `else` clause can assume that it will be executed only if no factor is found; if you don't hit the `break`, the number is prime.

The loop `else` clause is also run if the body of the loop is never executed, as you don't run a `break` in that event either; in a `while` loop, this happens if the test in the header is false to begin with. Thus, in the preceding example you still get the "is prime" message if `x` is initially less than or equal to 1 (for instance, if `y` is 2).

This example determines primes, but only informally so. Numbers less than 2 are not considered prime by the strict mathematical definition. To be really picky, this code also fails for negative numbers and succeeds for floating-point numbers with no decimal digits. Also note that its code must use `//` instead of `/` in Python 3.0 because of the migration of `/` to "true division," as described in Chapter 5 (we need the initial division to truncate remainders, not retain them!). If you want to experiment with this code, be sure to see the exercise at the end of Part IV, which wraps it in a function for reuse.

More on the loop else

Because the loop `else` clause is unique to Python, it tends to perplex some newcomers. In general terms, the loop `else` provides explicit syntax for a common coding scenario—it is a coding structure that lets us catch the "other" way out of a loop, without setting and checking flags or conditions.

Suppose, for instance, that we are writing a loop to search a list for a value, and we need to know whether the value was found after we exit the loop. We might code such a task this way:

```
found = False
while x and not found:
    if match(x[0]):                 # Value at front?
        print('Ni')
        found = True
    else:
        x = x[1:]                   # Slice off front and repeat
if not found:
    print('not found')
```

Here, we initialize, set, and later test a flag to determine whether the search succeeded or not. This is valid Python code, and it does work; however, this is exactly the sort of structure that the loop `else` clause is there to handle. Here's an `else` equivalent:

```
while x:                            # Exit when x empty
    if match(x[0]):
        print('Ni')
        break                       # Exit, go around else
    x = x[1:]
else:
    print('Not found')              # Only here if exhausted x
```

This version is more concise. The flag is gone, and we've replaced the `if` test at the loop end with an `else` (lined up vertically with the word `while`). Because the `break` inside the main part of the `while` exits the loop and goes around the `else`, this serves as a more structured way to catch the search-failure case.

Some readers might have noticed that the prior example's `else` clause could be replaced with a test for an empty `x` after the loop (e.g., `if not x:`). Although that's true in this example, the `else` provides explicit syntax for this coding pattern (it's more obviously a search-failure clause here), and such an explicit empty test may not apply in some cases. The loop `else` becomes even more useful when used in conjunction with the `for` loop—the topic of the next section—because sequence iteration is not under your control.

Why You Will Care: Emulating C while Loops

The section on expression statements in Chapter 11 stated that Python doesn't allow statements such as assignments to appear in places where it expects an expression. That means this common C language coding pattern won't work in Python:

```
while ((x = next()) != NULL) {...process x...}
```

C assignments return the value assigned, but Python assignments are just statements, not expressions. This eliminates a notorious class of C errors (you can't accidentally type = in Python when you mean ==). If you need similar behavior, though, there are at least three ways to get the same effect in Python `while` loops without embedding assignments in loop tests. You can move the assignment into the loop body with a `break`:

```
while True:
    x = next()
    if not x: break
    ...process x...
```

or move the assignment into the loop with tests:

```
x = True
while x:
    x = next()
    if x:
        ...process x...
```

or move the first assignment outside the loop:

```
x = next()
while x:
    ...process x...
    x = next()
```

Of these three coding patterns, the first may be considered by some to be the least structured, but it also seems to be the simplest and is the most commonly used. A simple Python `for` loop may replace some C loops as well.

for Loops

The `for` loop is a generic sequence iterator in Python: it can step through the items in any ordered sequence object. The `for` statement works on strings, lists, tuples, other built-in iterables, and new objects that we'll see how to create later with classes. We met it in brief when studying sequence object types; let's expand on its usage more formally here.

General Format

The Python `for` loop begins with a header line that specifies an assignment target (or targets), along with the object you want to step through. The header is followed by a block of (normally indented) statements that you want to repeat:

```
for <target> in <object>:              # Assign object items to target
    <statements>                       # Repeated loop body: use target
else:
    <statements>                       # If we didn't hit a 'break'
```

When Python runs a `for` loop, it assigns the items in the sequence object to the target one by one and executes the loop body for each. The loop body typically uses the assignment target to refer to the current item in the sequence as though it were a cursor stepping through the sequence.

The name used as the assignment target in a `for` header line is usually a (possibly new) variable in the scope where the `for` statement is coded. There's not much special about it; it can even be changed inside the loop's body, but it will automatically be set to the next item in the sequence when control returns to the top of the loop again. After the loop this variable normally still refers to the last item visited, which is the last item in the sequence unless the loop exits with a `break` statement.

The `for` statement also supports an optional `else` block, which works exactly as it does in a `while` loop—it's executed if the loop exits without running into a `break` statement (i.e., if all items in the sequence have been visited). The `break` and `continue` statements introduced earlier also work the same in a `for` loop as they do in a `while`. The `for` loop's complete format can be described this way:

```
for <target> in <object>:              # Assign object items to target
    <statements>
    if <test>: break                   # Exit loop now, skip else
    if <test>: continue                # Go to top of loop now
else:
    <statements>                       # If we didn't hit a 'break'
```

Examples

Let's type a few `for` loops interactively now, so you can see how they are used in practice.

Basic usage

As mentioned earlier, a `for` loop can step across any kind of sequence object. In our first example, for instance, we'll assign the name `x` to each of the three items in a list in turn, from left to right, and the `print` statement will be executed for each. Inside the `print` statement (the loop body), the name `x` refers to the current item in the list:

```
>>> for x in ["spam", "eggs", "ham"]:
...     print(x, end=' ')
...
spam eggs ham
```

The next two examples compute the sum and product of all the items in a list. Later in this chapter and later in the book we'll meet tools that apply operations such as `+` and `*` to items in a list automatically, but it's usually just as easy to use a `for`:

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item
...
>>> prod
24
```

Other data types

Any sequence works in a `for`, as it's a generic tool. For example, `for` loops work on strings and tuples:

```
>>> S = "lumberjack"
>>> T = ("and", "I'm", "okay")

>>> for x in S: print(x, end=' ')      # Iterate over a string
...
l u m b e r j a c k

>>> for x in T: print(x, end=' ')      # Iterate over a tuple
...
and I'm okay
```

In fact, as we'll in the next chapter when we explore the notion of "iterables," `for` loops can even work on some objects that are not sequences—files and dictionaries work, too!

Tuple assignment in for loops

If you're iterating through a sequence of tuples, the loop target itself can actually be a *tuple* of targets. This is just another case of the tuple-unpacking assignment we studied in Chapter 11 at work. Remember, the `for` loop assigns items in the sequence object to the target, and assignment works the same everywhere:

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:                   # Tuple assignment at work
...     print(a, b)
...
1 2
3 4
5 6
```

Here, the first time through the loop is like writing `(a,b) = (1,2)`, the second time is like writing `(a,b) = (3,4)`, and so on. The net effect is to automatically unpack the current tuple on each iteration.

This form is commonly used in conjunction with the `zip` call we'll meet later in this chapter to implement parallel traversals. It also makes regular appearances in conjunction with SQL databases in Python, where query result tables are returned as sequences

of sequences like the list used here—the outer list is the database table, the nested tuples are the rows within the table, and tuple assignment extracts columns.

Tuples in `for` loops also come in handy to iterate through *both* keys and values in dictionaries using the `items` method, rather than looping through the keys and indexing to fetch the values manually:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> for key in D:
...     print(key, '=>', D[key])              # Use dict keys iterator and index
...
a => 1
c => 3
b => 2

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (key, value) in D.items():
...     print(key, '=>', value)               # Iterate over both keys and values
...
a => 1
c => 3
b => 2
```

It's important to note that tuple assignment in `for` loops isn't a special case; any assignment target works syntactically after the word `for`. Although we can always assign manually within the loop to unpack:

```
>>> T
[(1, 2), (3, 4), (5, 6)]

>>> for both in T:
...     a, b = both                           # Manual assignment equivalent
...     print(a, b)
...
1 2
3 4
5 6
```

Tuples in the loop header save us an extra step when iterating through sequences of sequences. As suggested in Chapter 11, even *nested* structures may be automatically unpacked this way in a `for`:

```
>>> ((a, b), c) = ((1, 2), 3)               # Nested sequences work too
>>> a, b, c
(1, 2, 3)

>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)
...
1 2 3
4 5 6
```

But this is no special case—the `for` loop simply runs the sort of assignment we ran just before it, on each iteration. Any nested sequence structure may be unpacked this way, just because *sequence assignment* is so generic:

```
>>> for ((a, b), c) in [([1, 2], 3), ['XY', 6]]: print(a, b, c)
...
1 2 3
X Y 6
```

Python 3.0 extended sequence assignment in for loops

In fact, because the loop variable in a `for` loop can really be any assignment target, we can also use Python 3.0's extended sequence-unpacking assignment syntax here to extract items and sections of sequences within sequences. Really, this isn't a special case either, but simply a new assignment form in 3.0 (as discussed in Chapter 11); because it works in assignment statements, it automatically works in `for` loops.

Consider the tuple assignment form introduced in the prior section. A tuple of values is assigned to a tuple of names on each iteration, exactly like a simple assignment statement:

```
>>> a, b, c = (1, 2, 3)                    # Tuple assignment
>>> a, b, c
(1, 2, 3)

>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:    # Used in for loop
...     print(a, b, c)
...
1 2 3
4 5 6
```

In Python 3.0, because a sequence can be assigned to a more general set of names with a starred name to collect multiple items, we can use the same syntax to extract parts of nested sequences in the `for` loop:

```
>>> a, *b, c = (1, 2, 3, 4)                # Extended seq assignment
>>> a, b, c
(1, [2, 3], 4)

>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
...     print(a, b, c)
...
1 [2, 3] 4
5 [6, 7] 8
```

In practice, this approach might be used to pick out multiple columns from rows of data represented as nested sequences. In Python 2.X starred names aren't allowed, but you can achieve similar effects by slicing. The only difference is that slicing returns a type-specific result, whereas starred names always are assigned lists:

```
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:    # Manual slicing in 2.6
...     a, b, c = all[0], all[1:3], all[3]
...     print(a, b, c)
```

```
...
1 (2, 3) 4
5 (6, 7) 8
```

See Chapter 11 for more on this assignment form.

Nested for loops

Now let's look at a `for` loop that's a bit more sophisticated than those we've seen so far. The next example illustrates statement nesting and the loop `else` clause in a `for`. Given a list of objects (`items`) and a list of keys (`tests`), this code searches for each key in the objects list and reports on the search's outcome:

```
>>> items = ["aaa", 111, (4, 5), 2.01]   # A set of objects
>>> tests = [(4, 5), 3.14]               # Keys to search for
>>>
>>> for key in tests:                    # For all keys
...     for item in items:                # For all items
...         if item == key:               # Check for match
...             print(key, "was found")
...             break
...     else:
...         print(key, "not found!")
...
(4, 5) was found
3.14 not found!
```

Because the nested `if` runs a `break` when a match is found, the loop `else` clause can assume that if it is reached, the search has failed. Notice the nesting here. When this code runs, there are two loops going at the same time: the outer loop scans the keys list, and the inner loop scans the items list for each key. The nesting of the loop `else` clause is critical; it's indented to the same level as the header line of the inner `for` loop, so it's associated with the inner loop, not the `if` or the outer `for`.

Note that this example is easier to code if we employ the `in` operator to test membership. Because `in` implicitly scans an object looking for a match (at least logically), it replaces the inner loop:

```
>>> for key in tests:                    # For all keys
...     if key in items:                  # Let Python check for a match
...         print(key, "was found")
...     else:
...         print(key, "not found!")
...
(4, 5) was found
3.14 not found!
```

In general, it's a good idea to let Python do as much of the work as possible (as in this solution) for the sake of brevity and performance.

The next example performs a typical data-structure task with a `for`—collecting common items in two sequences (strings). It's roughly a simple set intersection routine; after the loop runs, `res` refers to a list that contains all the items found in `seq1` and `seq2`:

```
>>> seq1 = "spam"
>>> seq2 = "scam"
>>>
>>> res = []                          # Start empty
>>> for x in seq1:                    # Scan first sequence
...     if x in seq2:                 # Common item?
...         res.append(x)             # Add to result end
...
>>> res
['s', 'a', 'm']
```

Unfortunately, this code is equipped to work only on two specific variables: `seq1` and `seq2`. It would be nice if this loop could somehow be generalized into a tool you could use more than once. As you'll see, that simple idea leads us to *functions*, the topic of the next part of the book.

Why You Will Care: File Scanners

In general, loops come in handy anywhere you need to repeat an operation or process something more than once. Because files contain multiple characters and lines, they are one of the more typical use cases for loops. To load a file's contents into a string all at once, you simply call the file object's `read` method:

```
file = open('test.txt', 'r')    # Read contents into a string
print(file.read())
```

But to load a file in smaller pieces, it's common to code either a `while` loop with breaks on end-of-file, or a `for` loop. To read by characters, either of the following codings will suffice:

```
file = open('test.txt')
while True:
    char = file.read(1)          # Read by character
    if not char: break
    print(char)

for char in open('test.txt').read():
    print(char)
```

The `for` loop here also processes each character, but it loads the file into memory all at once (and assumes it fits!). To read by lines or blocks instead, you can use `while` loop code like this:

```
file = open('test.txt')
while True:
    line = file.readline()       # Read line by line
    if not line: break
    print(line, end='')          # Line already has a \n

file = open('test.txt', 'rb')
while True:
    chunk = file.read(10)        # Read byte chunks: up to 10 bytes
    if not chunk: break
    print(chunk)
```

You typically read binary data in blocks. To read text files line by line, though, the `for` loop tends to be easiest to code and the quickest to run:

```
for line in open('test.txt').readlines():
    print(line, end='')

for line in open('test.txt'):    # Use iterators: best text input mode
    print(line, end='')
```

The file `readlines` method loads a file all at once into a line-string list, and the last example here relies on file *iterators* to automatically read one line on each loop iteration (iterators are covered in detail in Chapter 14). See the library manual for more on the calls used here. The last example here is generally the best option for text files—besides its simplicity, it works for arbitrarily large files and doesn't load the entire file into memory all at once. The iterator version may be the quickest, but I/O performance is less clear-cut in Python 3.0.

In some 2.X Python code, you may also see the name `open` replaced with `file` and the file object's older `xreadlines` method used to achieve the same effect as the file's automatic line iterator (it's like `readlines` but doesn't load the file into memory all at once). Both `file` and `xreadlines` are removed in Python 3.0, because they are redundant; you shouldn't use them in 2.6 either, but they may pop up in older code and resources. Watch for more on reading files in Chapter 36; as we'll see there, text and binary files have slightly different semantics in 3.0.

Loop Coding Techniques

The `for` loop subsumes most counter-style loops. It's generally simpler to code and quicker to run than a `while`, so it's the first tool you should reach for whenever you need to step through a sequence. But there are also situations where you will need to iterate in more specialized ways. For example, what if you need to visit every second or third item in a list, or change the list along the way? How about traversing more than one sequence in parallel, in the same `for` loop?

You can always code such unique iterations with a `while` loop and manual indexing, but Python provides two built-ins that allow you to specialize the iteration in a `for`:

- The built-in `range` function produces a series of successively higher integers, which can be used as indexes in a `for`.
- The built-in `zip` function returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a `for`.

Because `for` loops typically run quicker than `while`-based counter loops, it's to your advantage to use tools like these that allow you to use `for` when possible. Let's look at each of these built-ins in turn.

Counter Loops: while and range

The `range` function is really a general tool that can be used in a variety of contexts. Although it's used most often to generate indexes in a `for`, you can use it anywhere you need a list of integers. In Python 3.0, `range` is an *iterator* that generates items on demand, so we need to wrap it in a `list` call to display its results all at once (more on iterators in Chapter 14):

```
>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
```

With one argument, `range` generates a list of integers from zero up to but not including the argument's value. If you pass in two arguments, the first is taken as the lower bound. An optional third argument can give a *step*; if it is used, Python adds the step to each successive integer in the result (the step defaults to 1). Ranges can also be nonpositive and nonascending, if you want them to be:

```
>>> list(range(-5, 5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(range(5, -5, -1))
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

Although such `range` results may be useful all by themselves, they tend to come in most handy within `for` loops. For one thing, they provide a simple way to repeat an action a specific number of times. To print three lines, for example, use a `range` to generate the appropriate number of integers; `for` loops force results from `range` automatically in 3.0, so we don't need `list` here:

```
>>> for i in range(3):
...     print(i, 'Pythons')
...
0 Pythons
1 Pythons
2 Pythons
```

`range` is also commonly used to iterate over a sequence indirectly. The easiest and fastest way to step through a sequence exhaustively is always with a simple `for`, as Python handles most of the details for you:

```
>>> X = 'spam'
>>> for item in X: print(item, end=' ')          # Simple iteration
...
s p a m
```

Internally, the `for` loop handles the details of the iteration automatically when used this way. If you really need to take over the indexing logic explicitly, you can do it with a `while` loop:

```
>>> i = 0
>>> while i < len(X):                           # while loop iteration
...     print(X[i], end=' ')
...     i += 1
```

```
...
s p a m
```

You can also do manual indexing with a `for`, though, if you use `range` to generate a list of indexes to iterate through. It's a multistep process, but it's sufficient to generate offsets, rather than the items at those offsets:

```
>>> X
'spam'
>>> len(X)                                  # Length of string
4
>>> list(range(len(X)))                     # All legal offsets into X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): print(X[i], end=' ')  # Manual for indexing
...
s p a m
```

Note that because this example is stepping over a list of *offsets* into `X`, not the actual *items* of `X`, we need to index back into `X` within the loop to fetch each item.

Nonexhaustive Traversals: range and Slices

The last example in the prior section works, but it's not the fastest option. It's also more work than we need to do. Unless you have a special indexing requirement, you're always better off using the simple `for` loop form in Python—as a general rule, use `for` instead of `while` whenever possible, and don't use `range` calls in `for` loops except as a last resort. This simpler solution is better:

```
>>> for item in X: print(item)              # Simple iteration
...
```

However, the coding pattern used in the prior example does allow us to do more specialized sorts of traversals. For instance, we can skip items as we go:

```
>>> S = 'abcdefghijk'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print(S[i], end=' ')
...
a c e g i k
```

Here, we visit every *second* item in the string `S` by stepping over the generated `range` list. To visit every third item, change the third `range` argument to be `3`, and so on. In effect, using `range` this way lets you skip items in loops while still retaining the simplicity of the `for` loop construct.

Still, this is probably not the ideal best-practice technique in Python today. If you really want to skip items in a sequence, the extended three-limit form of the *slice expression*, presented in Chapter 7, provides a simpler route to the same goal. To visit every second character in `S`, for example, slice with a stride of 2:

```
>>> S = 'abcdefghijk'
>>> for c in S[::2]: print(c, end=' ')
...
a c e g i k
```

The result is the same, but substantially easier for you to write and for others to read. The only real advantage to using `range` here instead is that it does not copy the string and does not create a list in 3.0; for very large strings, it may save memory.

Changing Lists: range

Another common place where you may use the `range` and `for` combination is in loops that change a list as it is being traversed. Suppose, for example, that you need to add 1 to every item in a list. You can try this with a simple `for` loop, but the result probably won't be exactly what you want:

```
>>> L = [1, 2, 3, 4, 5]

>>> for x in L:
...     x += 1
...
>>> L
[1, 2, 3, 4, 5]
>>> x
6
```

This doesn't quite work—it changes the loop variable `x`, not the list `L`. The reason is somewhat subtle. Each time through the loop, `x` refers to the next integer already pulled out of the list. In the first iteration, for example, `x` is integer `1`. In the next iteration, the loop body sets `x` to a different object, integer `2`, but it does not update the list where `1` originally came from.

To really change the list as we march across it, we need to use indexes so we can assign an updated value to each position as we go. The `range`/`len` combination can produce the required indexes for us:

```
>>> L = [1, 2, 3, 4, 5]

>>> for i in range(len(L)):          # Add one to each item in L
...     L[i] += 1                    # Or L[i] = L[i] + 1
...
>>> L
[2, 3, 4, 5, 6]
```

When coded this way, the list is changed as we proceed through the loop. There is no way to do the same with a simple `for x in L:`-style loop, because such a loop iterates through actual items, not list positions. But what about the equivalent `while` loop? Such a loop requires a bit more work on our part, and likely runs more slowly:

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
```

```
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

Here again, though, the `range` solution may not be ideal either. A list comprehension expression of the form:

```
[x+1 for x in L]
```

would do similar work, albeit without changing the original list in-place (we could assign the expression's new list object result back to `L`, but this would not update any other references to the original list). Because this is such a central looping concept, we'll save a complete exploration of list comprehensions for the next chapter.

Parallel Traversals: zip and map

As we've seen, the `range` built-in allows us to traverse sequences with `for` in a nonexhaustive fashion. In the same spirit, the built-in `zip` function allows us to use `for` loops to visit multiple sequences *in parallel*. In basic operation, `zip` takes one or more sequences as arguments and returns a series of tuples that pair up parallel items taken from those sequences. For example, suppose we're working with two lists:

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]
```

To combine the items in these lists, we can use `zip` to create a list of tuple pairs (like `range`, `zip` is an iterable object in 3.0, so we must wrap it in a `list` call to display all its results at once—more on iterators in the next chapter):

```
>>> zip(L1, L2)
<zip object at 0x026523C8>
>>> list(zip(L1, L2))                    # list() required in 3.0, not 2.6
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Such a result may be useful in other contexts as well, but when wedded with the `for` loop, it supports parallel iterations:

```
>>> for (x, y) in zip(L1, L2):
...     print(x, y, '--', x+y)
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

Here, we step over the result of the `zip` call—that is, the pairs of items pulled from the two lists. Notice that this `for` loop again uses the tuple assignment form we met earlier to unpack each tuple in the `zip` result. The first time through, it's as though we ran the assignment statement `(x, y) = (1, 5)`.

The net effect is that we scan both `L1` *and* `L2` in our loop. We could achieve a similar effect with a `while` loop that handles indexing manually, but it would require more typing and would likely run more slowly than the `for`/`zip` approach.

Strictly speaking, the `zip` function is more general than this example suggests. For instance, it accepts any type of sequence (really, any iterable object, including files), and it accepts more than two arguments. With three arguments, as in the following example, it builds a list of three-item tuples with items from each sequence, essentially projecting by columns (technically, we get an N-ary tuple for N arguments):

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> list(zip(T1, T2, T3))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Moreover, `zip` truncates result tuples at the length of the shortest sequence when the argument lengths differ. In the following, we zip together two strings to pick out characters in parallel, but the result has only as many tuples as the length of the shortest sequence:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> list(zip(S1, S2))
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

map equivalence in Python 2.6

In Python 2.X, the related built-in `map` function pairs items from sequences in a similar fashion, but it pads shorter sequences with `None` if the argument lengths differ instead of truncating to the shortest length:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'

>>> map(None, S1, S2)                          # 2.X only
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None,'3')]
```

This example is using a degenerate form of the `map` built-in, which is no longer supported in 3.0. Normally, `map` takes a function and one or more sequence arguments and collects the results of calling the function with parallel items taken from the sequence(s). We'll study `map` in detail in Chapters 19 and 20, but as a brief example, the following maps the built-in `ord` function across each item in a string and collects the results (like `zip`, `map` is a value generator in 3.0 and so must be passed to `list` to collect all its results at once):

```
>>> list(map(ord, 'spam'))
[115, 112, 97, 109]
```

This works the same as the following loop statement, but is often quicker:

```
>>> res = []
>>> for c in 'spam': res.append(ord(c))
>>> res
[115, 112, 97, 109]
```

Version skew note: The degenerate form of `map` using a function argument of `None` is no longer supported in Python 3.0, because it largely overlaps with `zip` (and was, frankly, a bit at odds with `map`'s function-application purpose). In 3.0, either use `zip` or write loop code to pad results yourself. We'll see how to do this in Chapter 20, after we've had a chance to study some additional iteration concepts.

Dictionary construction with zip

In Chapter 8, I suggested that the `zip` call used here can also be handy for generating dictionaries when the sets of keys and values must be computed at runtime. Now that we're becoming proficient with `zip`, I'll explain how it relates to dictionary construction. As you've learned, you can always create a dictionary by coding a dictionary literal, or by assigning to keys over time:

```
>>> D1 = {'spam':1, 'eggs':3, 'toast':5}
>>> D1
{'toast': 5, 'eggs': 3, 'spam': 1}

>>> D1 = {}
>>> D1['spam']  = 1
>>> D1['eggs']  = 3
>>> D1['toast'] = 5
```

What to do, though, if your program obtains dictionary keys and values in *lists* at runtime, after you've coded your script? For example, say you had the following keys and values lists:

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]
```

One solution for turning those lists into a dictionary would be to `zip` the lists and step through them in parallel with a `for` loop:

```
>>> list(zip(keys, vals))
[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'toast': 5, 'eggs': 3, 'spam': 1}
```

It turns out, though, that in Python 2.2 and later you can skip the `for` loop altogether and simply pass the zipped keys/values lists to the built-in `dict` constructor call:

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]

>>> D3 = dict(zip(keys, vals))
>>> D3
{'toast': 5, 'eggs': 3, 'spam': 1}
```

The built-in name `dict` is really a type name in Python (you'll learn more about type names, and subclassing them, in Chapter 31). Calling it achieves something like a list-to-dictionary conversion, but it's really an object construction request. In the next chapter we'll explore a related but richer concept, the *list comprehension*, which builds lists in a single expression; we'll also revisit 3.0 *dictionary comprehensions* an alternative to the `dict` cal for zipped key/value pairs.

Generating Both Offsets and Items: enumerate

Earlier, we discussed using `range` to generate the offsets of items in a string, rather than the items at those offsets. In some programs, though, we need both: the item to use, plus an offset as we go. Traditionally, this was coded with a simple `for` loop that also kept a counter of the current offset:

```
>>> S = 'spam'
>>> offset = 0
>>> for item in S:
...     print(item, 'appears at offset', offset)
...     offset += 1
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

This works, but in recent Python releases a new built-in named `enumerate` does the job for us:

```
>>> S = 'spam'
>>> for (offset, item) in enumerate(S):
...     print(item, 'appears at offset', offset)
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

The `enumerate` function returns a *generator object*—a kind of object that supports the iteration protocol that we will study in the next chapter and will discuss in more detail in the next part of the book. In short, it has a `__next__` method called by the `next` built-in function, which returns an (*`index, value`*) tuple each time through the loop. We can unpack these tuples with tuple assignment in the `for` loop (much like using `zip`):

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x02765AA8>
>>> next(E)
(0, 's')
>>> next(E)
(1, 'p')
>>> next(E)
(2, 'a')
```

As usual, we don't normally see this machinery because iteration contexts—including list comprehensions, the subject of Chapter 14—run the iteration protocol automatically:

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']
```

To fully understand iteration concepts like `enumerate`, `zip`, and list comprehensions, we need to move on to the next chapter for a more formal dissection.

Chapter Summary

In this chapter, we explored Python's looping statements as well as some concepts related to looping in Python. We looked at the `while` and `for` loop statements in depth, and we learned about their associated `else` clauses. We also studied the `break` and `continue` statements, which have meaning only inside loops, and met several built-in tools commonly used in `for` loops, including `range`, `zip`, `map`, and `enumerate` (although their roles as iterators in Python 3.0 won't be fully uncovered until the next chapter).

In the next chapter, we continue the iteration story by discussing list comprehensions and the iteration protocol in Python—concepts strongly related to `for` loops. There, we'll also explain some of the subtleties of iterable tools we met here, such as `range` and `zip`. As always, though, before moving on let's exercise what you've picked up here with a quiz.

Test Your Knowledge: Quiz

1. What are the main functional differences between a `while` and a `for`?
2. What's the difference between `break` and `continue`?
3. When is a loop's `else` clause executed?
4. How can you code a counter-based loop in Python?
5. What can a `range` be used for in a `for` loop?

Test Your Knowledge: Answers

1. The `while` loop is a general looping statement, but the `for` is designed to iterate across items in a sequence (really, iterable). Although the `while` can imitate the `for` with counter loops, it takes more code and might run slower.
2. The `break` statement exits a loop immediately (you wind up below the entire `while` or `for` loop statement), and `continue` jumps back to the top of the loop (you wind up positioned just before the test in `while` or the next item fetch in `for`).
3. The `else` clause in a `while` or `for` loop will be run once as the loop is exiting, if the loop exits normally (without running into a `break` statement). A `break` exits the loop immediately, skipping the `else` part on the way out (if there is one).
4. Counter loops can be coded with a `while` statement that keeps track of the index manually, or with a `for` loop that uses the `range` built-in function to generate successive integer offsets. Neither is the preferred way to work in Python, if you need to simply step across all the items in a sequence. Instead, use a simple `for` loop instead, without `range` or counters, whenever possible; it will be easier to code and usually quicker to run.
5. The `range` built-in can be used in a `for` to implement a fixed number of repetitions, to scan by offsets instead of items at offsets, to skip successive items as you go, and to change a list while stepping across it. None of these roles requires `range`, and most have alternatives—scanning actual items, three-limit slices, and list comprehensions are often better solutions today (despite the natural inclinations of ex-C programmers to want to count things!).

CHAPTER 14

Iterations and Comprehensions, Part 1

In the prior chapter we met Python's two looping statements, `while` and `for`. Although they can handle most repetitive tasks programs need to perform, the need to iterate over sequences is so common and pervasive that Python provides additional tools to make it simpler and more efficient. This chapter begins our exploration of these tools. Specifically, it presents the related concepts of Python's *iteration protocol*—a method-call model used by the `for` loop—and fills in some details on *list comprehensions*—a close cousin to the `for` loop that applies an expression to items in an iterable.

Because both of these tools are related to both the `for` loop and functions, we'll take a two-pass approach to covering them in this book: this chapter introduces the basics in the context of looping tools, serving as something of continuation of the prior chapter, and a later chapter (Chapter 20) revisits them in the context of function-based tools. In this chapter, we'll also sample additional iteration tools in Python and touch on the new iterators available in Python 3.0.

One note up front: some of the concepts presented in these chapters may seem advanced at first glance. With practice, though, you'll find that these tools are useful and powerful. Although never strictly required, because they've become commonplace in Python code, a basic understanding can also help if you must read programs written by others.

Iterators: A First Look

In the preceding chapter, I mentioned that the `for` loop can work on any sequence type in Python, including lists, tuples, and strings, like this:

```
>>> for x in [1, 2, 3, 4]: print(x ** 2, end=' ')
...
1 4 9 16

>>> for x in (1, 2, 3, 4): print(x ** 3, end=' ')
...
1 8 27 64
```

```
>>> for x in 'spam': print(x * 2, end=' ')
...
ss pp aa mm
```

Actually, the `for` loop turns out to be even more generic than this—it works on any *iterable object*. In fact, this is true of all iteration tools that scan objects from left to right in Python, including `for` loops, the list comprehensions we'll study in this chapter, `in` membership tests, the `map` built-in function, and more.

The concept of "iterable objects" is relatively recent in Python, but it has come to permeate the language's design. It's essentially a generalization of the notion of sequences—an object is considered *iterable* if it is either a physically stored sequence or an object that produces one result at a time in the context of an iteration tool like a `for` loop. In a sense, iterable objects include both physical sequences and *virtual sequences* computed on demand.*

The Iteration Protocol: File Iterators

One of the easiest ways to understand what this means is to look at how it works with a built-in type such as the file. Recall from Chapter 9 that open file objects have a method called `readline`, which reads one line of text from a file at a time—each time we call the `readline` method, we advance to the next line. At the end of the file, an empty string is returned, which we can detect to break out of the loop:

```
>>> f = open('script1.py')      # Read a 4-line script file in this directory
>>> f.readline()                # readline loads one line on each call
'import sys\n'
>>> f.readline()
'print(sys.path)\n'
>>> f.readline()
'x = 2\n'
>>> f.readline()
'print(2 ** 33)\n'
>>> f.readline()                # Returns empty string at end-of-file
''
```

However, files also have a method named `__next__` that has a nearly identical effect—it returns the next line from a file each time it is called. The only noticeable difference is that `__next__` raises a built-in `StopIteration` exception at end-of-file instead of returning an empty string:

```
>>> f = open('script1.py')      # __next__ loads one line on each call too
>>> f.__next__()                # But raises an exception at end-of-file
'import sys\n'
>>> f.__next__()
'print(sys.path)\n'
```

* Terminology in this topic tends to be a bit loose. This text uses the terms "iterable" and "iterator" interchangeably to refer to an object that supports iteration in general. Sometimes the term "iterable" refers to an object that supports `iter` and "iterator" refers to an object return by `iter` that supports `next(I)`, but that convention is not universal in either the Python world or this book.

```
>>> f.__next__()
'x = 2\n'
>>> f.__next__()
'print(2 ** 33)\n'
>>> f.__next__()
Traceback (most recent call last):
...more exception text omitted...
StopIteration
```

This interface is exactly what we call the *iteration protocol* in Python. Any object with a `__next__` method to advance to a next result, which raises `StopIteration` at the end of the series of results, is considered iterable in Python. Any such object may also be stepped through with a `for` loop or other iteration tool, because all iteration tools normally work internally by calling `__next__` on each iteration and catching the `StopIteration` exception to determine when to exit.

The net effect of this magic is that, as mentioned in Chapter 9, the best way to read a text file line by line today is to *not read it at all*—instead, allow the `for` loop to automatically call `__next__` to advance to the next line on each iteration. The file object's iterator will do the work of automatically loading lines as you go. The following, for example, reads a file line by line, printing the uppercase version of each line along the way, without ever explicitly reading from the file at all:

```
>>> for line in open('script1.py'):         # Use file iterators to read by lines
...     print(line.upper(), end='')         # Calls __next__, catches StopIteration
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 33)
```

Notice that the `print` uses `end=''` here to suppress adding a `\n`, because line strings already have one (without this, our output would be double-spaced). This is considered the best way to read text files line by line today, for three reasons: it's the simplest to code, might be the quickest to run, and is the best in terms of memory usage. The older, original way to achieve the same effect with a `for` loop is to call the file `readlines` method to load the file's content into memory as a list of line strings:

```
>>> for line in open('script1.py').readlines():
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 33)
```

This `readlines` technique still works, but it is not considered the best practice today and performs poorly in terms of memory usage. In fact, because this version really does load the entire file into memory all at once, it will not even work for files too big to fit into the memory space available on your computer. By contrast, because it reads one line at a time, the iterator-based version is immune to such memory-explosion issues.

The iterator version might run quicker too, though this can vary per release (Python 3.0 made this advantage less clear-cut by rewriting I/O libraries to support Unicode text and be less system-dependent).

As mentioned in the prior chapter's sidebar, "Why You Will Care: File Scanners" on page 340, it's also possible to read a file line by line with a `while` loop:

```
>>> f = open('script1.py')
>>> while True:
...     line = f.readline()
...     if not line: break
...     print(line.upper(), end='')
...
...same output...
```

However, this may run slower than the iterator-based `for` loop version, because iterators run at C language speed inside Python, whereas the `while` loop version runs Python byte code through the Python virtual machine. Any time we trade Python code for C code, speed tends to increase. This is not an absolute truth, though, especially in Python 3.0; we'll see timing techniques later in this book for measuring the relative speed of alternatives like these.

Manual Iteration: iter and next

To support manual iteration code (with less typing), Python 3.0 also provides a built-in function, `next`, that automatically calls an object's `__next__` method. Given an iterable object `X`, the call `next(X)` is the same as `X.__next__()`, but noticeably simpler. With files, for instance, either form may be used:

```
>>> f = open('script1.py')
>>> f.__next__()                 # Call iteration method directly
'import sys\n'
>>> f.__next__()
'print(sys.path)\n'

>>> f = open('script1.py')
>>> next(f)                      # next built-in calls __next__
'import sys\n'
>>> next(f)
'print(sys.path)\n'
```

Technically, there is one more piece to the iteration protocol. When the `for` loop begins, it obtains an iterator from the iterable object by passing it to the `iter` built-in function; the object returned by `iter` has the required `next` method. This becomes obvious if we look at how `for` loops internally process built-in sequence types such as lists:

```
>>> L = [1, 2, 3]
>>> I = iter(L)                  # Obtain an iterator object
>>> I.next()                     # Call next to advance to next item
1
>>> I.next()
2
```

```
>>> I.next()
3
>>> I.next()
Traceback (most recent call last):
...more omitted...
StopIteration
```

This initial step is not required for files, because a file object is its own iterator. That is, files have their own `__next__` method and so do not need to return a different object that does:

```
>>> f = open('script1.py')
>>> iter(f) is f
True
>>> f.__next__()
'import sys\n'
```

Lists, and many other built-in objects, are not their own iterators because they support multiple open iterations. For such objects, we must call `iter` to start iterating:

```
>>> L = [1, 2, 3]
>>> iter(L) is L
False
>>> L.__next__()
AttributeError: 'list' object has no attribute '__next__'

>>> I = iter(L)
>>> I.__next__()
1
>>> next(I)                 # Same as I.__next__()
2
```

Although Python iteration tools call these functions automatically, we can use them to apply the iteration protocol *manually*, too. The following interaction demonstrates the equivalence between automatic and manual iteration:†

```
>>> L = [1, 2, 3]
>>>
>>> for X in L:                 # Automatic iteration
...     print(X ** 2, end=' ')  # Obtains iter, calls __next__, catches exceptions
...
1 4 9

>>> I = iter(L)                 # Manual iteration: what for loops usually do
```

† Technically speaking, the `for` loop calls the internal equivalent of `I.__next__`, instead of the `next(I)` used here. There is rarely any difference between the two, but as we'll see in the next section, there are some built-in objects in 3.0 (such as `os.popen` results) that support the former and not the latter, but may be still be iterated across in `for` loops. Your manual iterations can generally use either call scheme. If you care for the full story, in 3.0 `os.popen` results have been reimplemented with the `subprocess` module and a wrapper class, whose `__getattr__` method is no longer called in 3.0 for implicit `__next__` fetches made by the `next` built-in, but is called for explicit fetches by name—a 3.0 change issue we'll confront in Chapters 37 and 38, which apparently burns some standard library code too! Also in 3.0, the related 2.6 calls `os.popen2/3/4` are no longer available; use `subprocess.Popen` with appropriate arguments instead (see the Python 3.0 library manual for the new required code).

```
>>> while True:
...     try:                        # try statement catches exceptions
...         X = next(I)             # Or call I.__next__
...     except StopIteration:
...         break
...     print(X ** 2, end=' ')
...
1 4 9
```

To understand this code, you need to know that `try` statements run an action and catch exceptions that occur while the action runs (we'll explore exceptions in depth in Part VII). I should also note that `for` loops and other iteration contexts can sometimes work differently for user-defined classes, repeatedly indexing an object instead of running the iteration protocol. We'll defer that story until we study class operator overloading in Chapter 29.

Version skew note: In Python 2.6, the iteration method is named `X.next()` instead of `X.__next__()`. For portability, the `next(X)` built-in function is available in Python 2.6 too (but not earlier), and calls 2.6's `X.next()` instead of 3.0's `X.__next__()`. Iteration works the same in 2.6 in all other ways, though; simply use `X.next()` or `next(X)` for manual iterations, instead of 3.0's `X.__next__()`. Prior to 2.6, use manual `X.next()` calls instead of `next(X)`.

Other Built-in Type Iterators

Besides files and physical sequences like lists, other types have useful iterators as well. The classic way to step through the keys of a *dictionary*, for example, is to request its keys list explicitly:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> for key in D.keys():
...     print(key, D[key])
...
a 1
c 3
b 2
```

In recent versions of Python, though, dictionaries have an iterator that automatically returns one key at a time in an iteration context:

```
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'c'
>>> next(I)
'b'
>>> next(I)
Traceback (most recent call last):
```

```
...more omitted...
StopIteration
```

The net effect is that we no longer need to call the `keys` method to step through dictionary keys—the `for` loop will use the iteration protocol to grab one key each time through:

```
>>> for key in D:
...     print(key, D[key])
...
a 1
c 3
b 2
```

We can't delve into their details here, but other Python object types also support the iterator protocol and thus may be used in `for` loops too. For instance, *shelves* (an access-by-key filesystem for Python objects) and the results from `os.popen` (a tool for reading the output of shell commands) are iterable as well:

```
>>> import os
>>> P = os.popen('dir')
>>> P.__next__()
' Volume in drive C is SQ004828V03\n'
>>> P.__next__()
' Volume Serial Number is 08BE-3CD4\n'
>>> next(P)
TypeError: _wrap_close object is not an iterator
```

Notice that `popen` objects support a `P.next()` method in Python 2.6. In 3.0, they support the `P.__next__()` method, but not the `next(P)` built-in; since the latter is defined to call the former, it's not clear if this behavior will endure in future releases (as described in an earlier footnote, this appears to be an implementation issue). This is only an issue for manual iteration, though; if you iterate over these objects automatically with `for` loops and other iteration contexts (described in the next sections), they return successive lines in either Python version.

The iteration protocol also is the reason that we've had to wrap some results in a `list` call to see their values all at once. Objects that are iterable return results one at a time, not in a physical list:

```
>>> R = range(5)
>>> R                           # Ranges are iterables in 3.0
range(0, 5)
>>> I = iter(R)                 # Use iteration protocol to produce results
>>> next(I)
0
>>> next(I)
1
>>> list(range(5))              # Or use list to collect all results at once
[0, 1, 2, 3, 4]
```

Now that you have a better understanding of this protocol, you should be able to see how it explains why the `enumerate` tool introduced in the prior chapter works the way it does:

```
>>> E = enumerate('spam')          # enumerate is an iterable too
>>> E
<enumerate object at 0x0253F508>
>>> I = iter(E)
>>> next(I)                        # Generate results with iteration protocol
(0, 's')
>>> next(I)                        # Or use list to force generation to run
(1, 'p')
>>> list(enumerate('spam'))
[(0, 's'), (1, 'p'), (2, 'a'), (3, 'm')]
```

We don't normally see this machinery because `for` loops run it for us automatically to step through results. In fact, everything that scans left-to-right in Python employs the iteration protocol in the same way—including the topic of the next section.

List Comprehensions: A First Look

Now that we've seen how the iteration protocol works, let's turn to a very common use case. Together with `for` loops, list comprehensions are one of the most prominent contexts in which the iteration protocol is applied.

In the previous chapter, we learned how to use `range` to change a list as we step across it:

```
>>> L = [1, 2, 3, 4, 5]

>>> for i in range(len(L)):
...     L[i] += 10
...
>>> L
[11, 12, 13, 14, 15]
```

This works, but as I mentioned there, it may not be the optimal "best-practice" approach in Python. Today, the list comprehension expression makes many such prior use cases obsolete. Here, for example, we can replace the loop with a single expression that produces the desired result list:

```
>>> L = [x + 10 for x in L]
>>> L
[21, 22, 23, 24, 25]
```

The net result is the same, but it requires less coding on our part and is likely to run substantially faster. The list comprehension isn't exactly the same as the `for` loop statement version because it makes a *new* list object (which might matter if there are multiple references to the original list), but it's close enough for most applications and is a common and convenient enough approach to merit a closer look here.

List Comprehension Basics

We met the list comprehension briefly in Chapter 4. Syntactically, its syntax is derived from a construct in set theory notation that applies an operation to each item in a set, but you don't have to know set theory to use this tool. In Python, most people find that a list comprehension simply looks like a backward `for` loop.

To get a handle on the syntax, let's dissect the prior section's example in more detail:

```
>>> L = [x + 10 for x in L]
```

List comprehensions are written in square brackets because they are ultimately a way to construct a new list. They begin with an arbitrary expression that we make up, which uses a loop variable that we make up (`x + 10`). That is followed by what you should now recognize as the header of a `for` loop, which names the loop variable, and an iterable object (`for x in L`).

To run the expression, Python executes an iteration across `L` inside the interpreter, assigning `x` to each item in turn, and collects the results of running the items through the expression on the left side. The result list we get back is exactly what the list comprehension says—a new list containing `x + 10`, for every `x` in `L`.

Technically speaking, list comprehensions are never really required because we can always build up a list of expression results manually with `for` loops that append results as we go:

```
>>> res = []
>>> for x in L:
...     res.append(x + 10)
...
>>> res
[21, 22, 23, 24, 25]
```

In fact, this is exactly what the list comprehension does internally.

However, list comprehensions are more concise to write, and because this code pattern of building up result lists is so common in Python work, they turn out to be very handy in many contexts. Moreover, list comprehensions can run much faster than manual `for` loop statements (often roughly twice as fast) because their iterations are performed at C language speed inside the interpreter, rather than with manual Python code; especially for larger data sets, there is a major performance advantage to using them.

Using List Comprehensions on Files

Let's work through another common use case for list comprehensions to explore them in more detail. Recall that the file object has a `readlines` method that loads the file into a list of line strings all at once:

```
>>> f = open('script1.py')
>>> lines = f.readlines()
```

```
>>> lines
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']
```

This works, but the lines in the result all include the newline character (`\n`) at the end. For many programs, the newline character gets in the way—we have to be careful to avoid double-spacing when printing, and so on. It would be nice if we could get rid of these newlines all at once, wouldn't it?

Any time we start thinking about performing an operation on each item in a sequence, we're in the realm of list comprehensions. For example, assuming the variable `lines` is as it was in the prior interaction, the following code does the job by running each line in the list through the string `rstrip` method to remove whitespace on the right side (a `line[:-1]` slice would work, too, but only if we can be sure all lines are properly terminated):

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```

This works as planned. Because list comprehensions are an iteration context just like `for` loop statements, though, we don't even have to open the file ahead of time. If we open it inside the expression, the list comprehension will automatically use the iteration protocol we met earlier in this chapter. That is, it will read one line from the file at a time by calling the file's `next` method, run the line through the `rstrip` expression, and add it to the result list. Again, we get what we ask for—the `rstrip` result of a line, for every line in the file:

```
>>> lines = [line.rstrip() for line in open('script1.py')]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(2 ** 33)']
```

This expression does a lot implicitly, but we're getting a lot of work for free here—Python scans the file and builds a list of operation results automatically. It's also an efficient way to code this operation: because most of this work is done inside the Python interpreter, it is likely much faster than an equivalent `for` statement. Again, especially for large files, the speed advantages of list comprehensions can be significant.

Besides their efficiency, list comprehensions are also remarkably expressive. In our example, we can run any string operation on a file's lines as we iterate. Here's the list comprehension equivalent to the file iterator uppercase example we met earlier, along with a few others (the method chaining in the second of these examples works because string methods return a new string, to which we can apply another string method):

```
>>> [line.upper() for line in open('script1.py')]
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']

>>> [line.rstrip().upper() for line in open('script1.py')]
['IMPORT SYS', 'PRINT(SYS.PATH)', 'X = 2', 'PRINT(2 ** 33)']

>>> [line.split() for line in open('script1.py')]
[['import', 'sys'], ['print(sys.path)'], ['x', '=', '2'], ['print(2', '**','33)']]
```

```
>>> [line.replace(' ', '!') for line in open('script1.py')]
['import!sys\n', 'print(sys.path)\n', 'x!=!2\n', 'print(2!**!33)\n']

>>> [('sys' in line, line[0]) for line in open('script1.py')]
[(True, 'i'), (True, 'p'), (False, 'x'), (False, 'p')]
```

Extended List Comprehension Syntax

In fact, list comprehensions can be even more advanced in practice. As one particularly useful extension, the `for` loop nested in the expression can have an associated `if` clause to filter out of the result items for which the test is not true.

For example, suppose we want to repeat the prior section's file-scanning example, but we need to collect only lines that begin with the letter *p* (perhaps the first character on each line is an action code of some sort). Adding an `if` filter clause to our expression does the trick:

```
>>> lines = [line.rstrip() for line in open('script1.py') if line[0] == 'p']
>>> lines
['print(sys.path)', 'print(2 ** 33)']
```

Here, the `if` clause checks each line read from the file to see whether its first character is *p*; if not, the line is omitted from the result list. This is a fairly big expression, but it's easy to understand if we translate it to its simple `for` loop statement equivalent. In general, we can always translate a list comprehension to a `for` statement by appending as we go and further indenting each successive part:

```
>>> res = []
>>> for line in open('script1.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
...
>>> res
['print(sys.path)', 'print(2 ** 33)']
```

This `for` statement equivalent works, but it takes up four lines instead of one and probably runs substantially slower.

List comprehensions can become even more complex if we need them to—for instance, they may contain nested loops, coded as a series of `for` clauses. In fact, their full syntax allows for any number of `for` clauses, each of which can have an optional associated `if` clause (we'll be more formal about their syntax in Chapter 20).

For example, the following builds a list of the concatenation of `x + y` for every `x` in one string and every `y` in another. It effectively collects the permutation of the characters in two strings:

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Again, one way to understand this expression is to convert it to statement form by indenting its parts. The following is an equivalent, but likely slower, alternative way to achieve the same effect:

```
>>> res = []
>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
...
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Beyond this complexity level, though, list comprehension expressions can often become too compact for their own good. In general, they are intended for simple types of iterations; for more involved work, a simpler `for` statement structure will probably be easier to understand and modify in the future. As usual in programming, if something is difficult for you to understand, it's probably not a good idea.

We'll revisit list comprehensions in Chapter 20, in the context of functional programming tools; as we'll see, they turn out to be just as related to functions as they are to looping statements.

Other Iteration Contexts

Later in the book, we'll see that user-defined classes can implement the iteration protocol too. Because of this, it's sometimes important to know which built-in tools make use of it—any tool that employs the iteration protocol will automatically work on any built-in type or user-defined class that provides it.

So far, I've been demonstrating iterators in the context of the `for` loop statement, because this part of the book is focused on statements. Keep in mind, though, that every tool that scans from left to right across objects uses the iteration protocol. This includes the `for` loops we've seen:

```
>>> for line in open('script1.py'):          # Use file iterators
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 33)
```

However, list comprehensions, the `in` membership test, the `map` built-in function, and other built-ins such as the `sorted` and `zip` calls also leverage the iteration protocol. When applied to a file, all of these use the file object's iterator automatically to scan line by line:

```
>>> uppers = [line.upper() for line in open('script1.py')]
>>> uppers
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']
```

```
>>> map(str.upper, open('script1.py'))        # map is an iterable in 3.0
<map object at 0x02660710>

>>> list( map(str.upper, open('script1.py')) )
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']

>>> 'y = 2\n' in open('script1.py')
False
>>> 'x = 2\n' in open('script1.py')
True
```

We introduced the `map` call used here in the preceding chapter; it's a built-in that applies a function call to each item in the passed-in iterable object. `map` is similar to a list comprehension but is more limited because it requires a function instead of an arbitrary expression. It also *returns* an iterable object itself in Python 3.0, so we must wrap it in a `list` call to force it to give us all its values at once; more on this change later in this chapter. Because `map`, like the list comprehension, is related to both `for` loops and functions, we'll also explore both again in Chapters 19 and 20.

Python includes various additional built-ins that process iterables, too: `sorted` sorts items in an iterable, `zip` combines items from iterables, `enumerate` pairs items in an iterable with relative positions, `filter` selects items for which a function is true, and `reduce` runs pairs of items in an iterable through a function. All of these accept iterables, and `zip`, `enumerate`, and `filter` also return an iterable in Python 3.0, like `map`. Here they are in action running the file's iterator automatically to scan line by line:

```
>>> sorted(open('script1.py'))
['import sys\n', 'print(2 ** 33)\n', 'print(sys.path)\n', 'x = 2\n']

>>> list(zip(open('script1.py'), open('script1.py')))
[('import sys\n', 'import sys\n'), ('print(sys.path)\n', 'print(sys.path)\n'),
('x = 2\n', 'x = 2\n'), ('print(2 ** 33)\n', 'print(2 ** 33)\n')]

>>> list(enumerate(open('script1.py')))
[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'),
(3, 'print(2 ** 33)\n')]

>>> list(filter(bool, open('script1.py')))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']

>>> import functools, operator
>>> functools.reduce(operator.add, open('script1.py'))
'import sys\nprint(sys.path)\nx = 2\nprint(2 ** 33)\n'
```

All of these are iteration tools, but they have unique roles. We met `zip` and `enumerate` in the prior chapter; `filter` and `reduce` are in Chapter 19's functional programming domain, so we'll defer details for now.

We first saw the `sorted` function used here at work in Chapter 4, and we used it for dictionaries in Chapter 8. `sorted` is a built-in that employs the iteration protocol—it's like the original list `sort` method, but it returns the new sorted list as a result and runs

on any iterable object. Notice that, unlike `map` and others, `sorted` returns an actual *list* in Python 3.0 instead of an iterable.

Other built-in functions support the iteration protocol as well (but frankly, are harder to cast in interesting examples related to files). For example, the `sum` call computes the sum of all the numbers in any iterable; the `any` and `all` built-ins return `True` if any or all items in an iterable are `True`, respectively; and `max` and `min` return the largest and smallest item in an iterable, respectively. Like `reduce`, all of the tools in the following examples accept any iterable as an argument and use the iteration protocol to scan it, but return a single result:

```
>>> sum([3, 2, 4, 1, 5, 0])                # sum expects numbers only
15
>>> any(['spam', '', 'ni'])
True
>>> all(['spam', '', 'ni'])
False
>>> max([3, 2, 5, 1, 4])
5
>>> min([3, 2, 5, 1, 4])
1
```

Strictly speaking, the `max` and `min` functions can be applied to files as well—they automatically use the iteration protocol to scan the file and pick out the lines with the highest and lowest string values, respectively (though I'll leave valid use cases to your imagination):

```
>>> max(open('script1.py'))                 # Line with max/min string value
'x = 2\n'
>>> min(open('script1.py'))
'import sys\n'
```

Interestingly, the iteration protocol is even more pervasive in Python today than the examples so far have demonstrated—*everything* in Python's built-in toolset that scans an object from left to right is defined to use the iteration protocol on the subject object. This even includes more esoteric tools such as the `list` and `tuple` built-in functions (which build new objects from iterables), the string `join` method (which puts a substring between strings contained in an iterable), and even sequence assignments. Consequently, all of these will also work on an open file and automatically read one line at a time:

```
>>> list(open('script1.py'))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']

>>> tuple(open('script1.py'))
('import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n')

>>> '&&'.join(open('script1.py'))
'import sys\n&&print(sys.path)\n&&x = 2\n&&print(2 ** 33)\n'

>>> a, b, c, d = open('script1.py')
>>> a, d
```

```
('import sys\n', 'print(2 ** 33)\n')

>>> a, *b = open('script1.py')                  # 3.0 extended form
>>> a, b
('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n'])
```

Earlier, we saw that the built-in `dict` call accepts an iterable `zip` result, too. For that matter, so does the `set` call, as well as the new *set* and *dictionary comprehension expressions* in Python 3.0, which we met in Chapters 4, 5, and 8:

```
>>> set(open('script1.py'))
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}

>>> {line for line in open('script1.py')}
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}

>>> {ix: line for ix, line in enumerate(open('script1.py'))}
{0: 'import sys\n', 1: 'print(sys.path)\n', 2: 'x = 2\n', 3: 'print(2 ** 33)\n'}
```

In fact, both set and dictionary comprehensions support the extended syntax of list comprehensions we met earlier in this chapter, including `if` tests:

```
>>> {line for line in open('script1.py') if line[0] == 'p'}
{'print(sys.path)\n', 'print(2 ** 33)\n'}

>>> {ix: line for (ix, line) in enumerate(open('script1.py')) if line[0] == 'p'}
{1: 'print(sys.path)\n', 3: 'print(2 ** 33)\n'}
```

Like the list comprehension, both of these scan the file line by line and pick out lines that begin with the letter "p." They also happen to build sets and dictionaries in the end, but we get a lot of work "for free" by combining file iteration and comprehension syntax.

There's one last iteration context that's worth mentioning, although it's a bit of a preview: in Chapter 18, we'll learn that a special `*arg` form can be used in function calls to unpack a collection of values into individual arguments. As you can probably predict by now, this accepts any iterable, too, including files (see Chapter 18 for more details on the call syntax):

```
>>> def f(a, b, c, d): print(a, b, c, d, sep='&')
...
>>> f(1, 2, 3, 4)
1&2&3&4
>>> f(*[1, 2, 3, 4])                    # Unpacks into arguments
1&2&3&4

>>> f(*open('script1.py'))              # Iterates by lines too!
import sys
&print(sys.path)
&x = 2
&print(2 ** 33)
```

In fact, because this argument-unpacking syntax in calls accepts iterables, it's also possible to use the `zip` built-in to *unzip* zipped tuples, by making prior or nested `zip` results

arguments for another `zip` call (warning: you probably shouldn't read the following example if you plan to operate heavy machinery anytime soon!):

```
>>> X = (1, 2)
>>> Y = (3, 4)
>>>
>>> list(zip(X, Y))                    # Zip tuples: returns an iterable
[(1, 3), (2, 4)]
>>>
>>> A, B = zip(*zip(X, Y))             # Unzip a zip!
>>> A
(1, 2)
>>> B
(3, 4)
```

Still other tools in Python, such as the `range` built-in and dictionary view objects, return iterables instead of processing them. To see how these have been absorbed into the iteration protocol in Python 3.0 as well, we need to move on to the next section.

New Iterables in Python 3.0

One of the fundamental changes in Python 3.0 is that it has a stronger emphasis on iterators than 2.X. In addition to the iterators associated with built-in types such as files and dictionaries, the dictionary methods `keys`, `values`, and `items` return iterable objects in Python 3.0, as do the built-in functions `range`, `map`, `zip`, and `filter`. As shown in the prior section, the last three of these functions both return iterators and process them. All of these tools produce results on demand in Python 3.0, instead of constructing result lists as they do in 2.6.

Although this saves memory space, it can impact your coding styles in some contexts. In various places in this book so far, for example, we've had to wrap up various function and method call results in a `list(...)` call in order to force them to produce all their results at once:

```
>>> zip('abc', 'xyz')                  # An iterable in Python 3.0 (a list in 2.6)
<zip object at 0x02E66710>

>>> list(zip('abc', 'xyz'))            # Force list of results in 3.0 to display
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

This isn't required in 2.6, because functions like `zip` return lists of results. In 3.0, though, they return iterable objects, producing results on demand. This means extra typing is required to display the results at the interactive prompt (and possibly in some other contexts), but it's an asset in larger programs—delayed evaluation like this conserves memory and avoids pauses while large result lists are computed. Let's take a quick look at some of the new 3.0 iterables in action.

The range Iterator

We studied the `range` built-in's basic behavior in the prior chapter. In 3.0, it returns an iterator that generates numbers in the range on demand, instead of building the result list in memory. This subsumes the older 2.X `xrange` (see the upcoming version skew note), and you must use `list(range(...))` to force an actual range list if one is needed (e.g., to display results):

```
C:\\misc> c:\python30\python
>>> R = range(10)                  # range returns an iterator, not a list
>>> R
range(0, 10)

>>> I = iter(R)                    # Make an iterator from the range
>>> next(I)                        # Advance to next result
0                                  # What happens in for loops, comprehensions, etc.
>>> next(I)
1
>>> next(I)
2

>>> list(range(10))                # To force a list if required
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Unlike the list returned by this call in 2.X, `range` objects in 3.0 support only iteration, indexing, and the `len` function. They do not support any other sequence operations (use `list(...)` if you require more list tools):

```
>>> len(R)                         # range also does len and indexing, but no others
10
>>> R[0]
0
>>> R[-1]
9

>>> next(I)                        # Continue taking from iterator, where left off
3
>>> I.__next__()                   # .next() becomes .__next__(), but use new next()
4
```

Version skew note: Python 2.X also has a built-in called `xrange`, which is like `range` but produces items on demand instead of building a list of results in memory all at once. Since this is exactly what the new iterator-based `range` does in Python 3.0, `xrange` is no longer available in 3.0—it has been subsumed. You may still see it in 2.X code, though, especially since `range` builds result lists there and so is not as efficient in its memory usage. As noted in a sidebar in the prior chapter, the `file.xreadlines()` method used to minimize memory use in 2.X has been dropped in Python 3.0 for similar reasons, in favor of file iterators.

The map, zip, and filter Iterators

Like `range`, the `map`, `zip`, and `filter` built-ins also become iterators in 3.0 to conserve space, rather than producing a result list all at once in memory. All three not only process iterables, as in 2.X, but also return iterable results in 3.0. Unlike `range`, though, they are their own iterators—after you step through their results once, they are exhausted. In other words, you can't have multiple iterators on their results that maintain different positions in those results.

Here is the case for the `map` built-in we met in the prior chapter. As with other iterators, you can force a list with `list(...)` if you really need one, but the default behavior can save substantial space in memory for large result sets:

```
>>> M = map(abs, (-1, 0, 1))            # map returns an iterator, not a list
>>> M
<map object at 0x0276B890>
>>> next(M)                             # Use iterator manually: exhausts results
1                                       # These do not support len() or indexing
>>> next(M)
0
>>> next(M)
1
>>> next(M)
StopIteration

>>> for x in M: print(x)                # map iterator is now empty: one pass only
...

>>> M = map(abs, (-1, 0, 1))            # Make a new iterator to scan again
>>> for x in M: print(x)                # Iteration contexts auto call next()
...
1
0
1
>>> list(map(abs, (-1, 0, 1)))          # Can force a real list if needed
[1, 0, 1]
```

The `zip` built-in, introduced in the prior chapter, returns iterators that work the same way:

```
>>> Z = zip((1, 2, 3), (10, 20, 30))    # zip is the same: a one-pass iterator
>>> Z
<zip object at 0x02770EE0>

>>> list(Z)
[(1, 10), (2, 20), (3, 30)]

>>> for pair in Z: print(pair)          # Exhausted after one pass
...

>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> for pair in Z: print(pair)          # Iterator used automatically or manually
...
(1, 10)
```

```
(2, 20)
(3, 30)

>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> next(Z)
(1, 10)
>>> next(Z)
(2, 20)
```

The `filter` built-in, which we'll study in the next part of this book, is also analogous. It returns items in an iterable for which a passed-in function returns `True` (as we've learned, in Python `True` includes nonempty objects):

```
>>> filter(bool, ['spam', '', 'ni'])
<filter object at 0x0269C6D0>
>>> list(filter(bool, ['spam', '', 'ni']))
['spam', 'ni']
```

Like most of the tools discussed in this section, `filter` both accepts an iterable to process and returns an iterable to generate results in 3.0.

Multiple Versus Single Iterators

It's interesting to see how the `range` object differs from the built-ins described in this section—it supports `len` and indexing, it is not its own iterator (you make one with `iter` when iterating manually), and it supports multiple iterators over its result that remember their positions independently:

```
>>> R = range(3)                        # range allows multiple iterators
>>> next(R)
TypeError: range object is not an iterator

>>> I1 = iter(R)
>>> next(I1)
0
>>> next(I1)
1
>>> I2 = iter(R)                        # Two iterators on one range
>>> next(I2)
0
>>> next(I1)                            # I1 is at a different spot than I2
2
```

By contrast, `zip`, `map`, and `filter` do not support multiple active iterators on the same result:

```
>>> Z = zip((1, 2, 3), (10, 11, 12))
>>> I1 = iter(Z)
>>> I2 = iter(Z)                        # Two iterators on one zip
>>> next(I1)
(1, 10)
>>> next(I1)
(2, 11)
>>> next(I2)                            # I2 is at same spot as I1!
```

```
(3, 12)

>>> M = map(abs, (-1, 0, 1))                # Ditto for map (and filter)
>>> I1 = iter(M); I2 = iter(M)
>>> print(next(I1), next(I1), next(I1))
1 0 1
>>> next(I2)
StopIteration

>>> R = range(3)                            # But range allows many iterators
>>> I1, I2 = iter(R), iter(R)
>>> [next(I1), next(I1), next(I1)]
[0 1 2]
>>> next(I2)
0
```

When we code our own iterable objects with classes later in the book (Chapter 29), we'll see that multiple iterators are usually supported by returning new objects for the `iter` call; a single iterator generally means an object returns itself. In Chapter 20, we'll also find that *generator functions and expressions* behave like `map` and `zip` instead of `range` in this regard, supporting a single active iteration. In that chapter, we'll see some subtle implications of one-shot iterators in loops that attempt to scan multiple times.

Dictionary View Iterators

As we saw briefly in Chapter 8, in Python 3.0 the dictionary `keys`, `values`, and `items` methods return iterable *view* objects that generate result items one at a time, instead of producing result lists all at once in memory. View items maintain the same physical ordering as that of the dictionary and reflect changes made to the underlying dictionary. Now that we know more about iterators, here's the rest of the story:

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> K = D.keys()                            # A view object in 3.0, not a list
>>> K
<dict_keys object at 0x026D83C0>

>>> next(K)                                 # Views are not iterators themselves
TypeError: dict_keys object is not an iterator

>>> I = iter(K)                             # Views have an iterator,
>>> next(I)                                 # which can be used manually
'a'                                         # but does not support len(), index
>>> next(I)
'c'

>>> for k in D.keys(): print(k, end=' ')    # All iteration contexts use auto
...
a c b
```

As for all iterators, you can always force a 3.0 dictionary view to build a real list by passing it to the `list` built-in. However, this usually isn't required except to display results interactively or to apply list operations like indexing:

```
>>> K = D.keys()
>>> list(K)                              # Can still force a real list if needed
['a', 'c', 'b']

>>> V = D.values()                       # Ditto for values() and items() views
>>> V
<dict_values object at 0x026D8260>
>>> list(V)
[1, 3, 2]

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (k, v) in D.items(): print(k, v, end=' ')
...
a 1 c 3 b 2
```

In addition, 3.0 dictionaries still have iterators themselves, which return successive keys. Thus, it's not often necessary to call `keys` directly in this context:

```
>>> D                                    # Dictionaries still have own iterator
{'a': 1, 'c': 3, 'b': 2}                 # Returns next key on each iteration
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'c'

>>> for key in D: print(key, end=' ')    # Still no need to call keys() to iterate
...                                      # But keys is an iterator in 3.0 too!
a c b
```

Finally, remember again that because `keys` no longer returns a list, the traditional coding pattern for scanning a dictionary by sorted keys won't work in 3.0. Instead, convert keys views first with a `list` call, or use the `sorted` call on either a keys view or the dictionary itself, as follows:

```
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> for k in sorted(D.keys()): print(k, D[k], end=' ')
...
a 1 b 2 c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> for k in sorted(D): print(k, D[k], end=' ')     # Best practice key sorting
...
a 1 b 2 c 3
```

Other Iterator Topics

We'll learn more about both list comprehensions and iterators in Chapter 20, in conjunction with functions, and again in Chapter 29 when we study classes. As you'll see later:

- User-defined functions can be turned into iterable *generator functions*, with `yield` statements.
- List comprehensions morph into iterable *generator expressions* when coded in parentheses.
- User-defined classes are made iterable with `__iter__` or `__getitem__` *operator overloading*.

In particular, user-defined iterators defined with classes allow arbitrary objects and operations to be used in any of the iteration contexts we've met here.

Chapter Summary

In this chapter, we explored concepts related to looping in Python. We took our first substantial look at the *iteration protocol* in Python—a way for nonsequence objects to take part in iteration loops—and at *list comprehensions*. As we saw, a list comprehension is an expression similar to a `for` loop that applies another expression to all the items in any iterable object. Along the way, we also saw other built-in iteration tools at work and studied recent iteration additions in Python 3.0.

This wraps up our tour of specific procedural statements and related tools. The next chapter closes out this part of the book by discussing documentation options for Python code; documentation is also part of the general syntax model, and it's an important component of well-written programs. In the next chapter, we'll also dig into a set of exercises for this part of the book before we turn our attention to larger structures such as functions. As usual, though, let's first exercise what we've learned here with a quiz.

Test Your Knowledge: Quiz

1. How are `for` loops and iterators related?
2. How are `for` loops and list comprehensions related?
3. Name four iteration contexts in the Python language.
4. What is the best way to read line by line from a text file today?
5. What sort of weapons would you expect to see employed by the Spanish Inquisition?

Test Your Knowledge: Answers

1. The `for` loop uses the *iteration protocol* to step through items in the object across which it is iterating. It calls the object's `__next__` method (run by the `next` built-in) on each iteration and catches the `StopIteration` exception to determine when to stop looping. Any object that supports this model works in a `for` loop and in other iteration contexts.
2. Both are iteration tools. List comprehensions are a concise and efficient way to perform a common `for` loop task: collecting the results of applying an expression to all items in an iterable object. It's always possible to translate a list comprehension to a `for` loop, and part of the list comprehension expression looks like the header of a `for` loop syntactically.
3. Iteration contexts in Python include the `for` loop; list comprehensions; the `map` built-in function; the `in` membership test expression; and the built-in functions `sorted`, `sum`, `any`, and `all`. This category also includes the `list` and `tuple` built-ins, string `join` methods, and sequence assignments, all of which use the iteration protocol (the `__next__` method) to step across iterable objects one item at a time.
4. The best way to read lines from a text file today is to not read it explicitly at all: instead, open the file within an iteration context such as a `for` loop or list comprehension, and let the iteration tool automatically scan one line at a time by running the file's `next` method on each iteration. This approach is generally best in terms of coding simplicity, execution speed, and memory space requirements.
5. I'll accept any of the following as correct answers: fear, intimidation, nice red uniforms, a comfy chair, and soft pillows.

CHAPTER 15

The Documentation Interlude

This part of the book concludes with a look at techniques and tools used for documenting Python code. Although Python code is designed to be readable, a few well-placed human-readable comments can do much to help others understand the workings of your programs. Python includes syntax and tools to make documentation easier.

Although this is something of a tools-related concept, the topic is presented here partly because it involves Python's syntax model, and partly as a resource for readers struggling to understand Python's toolset. For the latter purpose, I'll expand here on documentation pointers first given in Chapter 4. As usual, in addition to the chapter quiz this concluding chapter ends with some warnings about common pitfalls and a set of exercises for this part of the text.

Python Documentation Sources

By this point in the book, you're probably starting to realize that Python comes with an amazing amount of prebuilt functionality—built-in functions and exceptions, predefined object attributes and methods, standard library modules, and more. And we've really only scratched the surface of each of these categories.

One of the first questions that bewildered beginners often ask is: how do I find information on all the built-in tools? This section provides hints on the various documentation sources available in Python. It also presents *documentation strings* (docstrings) and the *PyDoc* system that makes use of them. These topics are somewhat peripheral to the core language itself, but they become essential knowledge as soon as your code reaches the level of the examples and exercises in this part of the book.

As summarized in Table 15-1, there are a variety of places to look for information on Python, with generally increasing verbosity. Because documentation is such a crucial tool in practical programming, we'll explore each of these categories in the sections that follow.

Table 15-1. Python documentation sources

Form	Role
# comments	In-file documentation
The `dir` function	Lists of attributes available in objects
Docstrings: `__doc__`	In-file documentation attached to objects
PyDoc: The `help` function	Interactive help for objects
PyDoc: HTML reports	Module documentation in a browser
The standard manual set	Official language and library descriptions
Web resources	Online tutorials, examples, and so on
Published books	Commercially available reference texts

Comments

Hash-mark comments are the most basic way to document your code. Python simply ignores all the text following a `#` (as long as it's not inside a string literal), so you can follow this character with words and descriptions meaningful to programmers. Such comments are accessible only in your source files, though; to code comments that are more widely available, you'll need to use docstrings.

In fact, current best practice generally dictates that docstrings are best for larger functional documentation (e.g., "my file does this"), and `#` comments are best limited to smaller code documentation (e.g., "this strange expression does that"). More on docstrings in a moment.

The dir Function

The built-in `dir` function is an easy way to grab a list of all the attributes available inside an object (i.e., its methods and simpler data items). It can be called on any object that has attributes. For example, to find out what's available in the standard library's `sys` module, import it and pass it to `dir` (these results are from Python 3.0; they might vary slightly on 2.6):

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
'_getframe', 'api_version', 'argv', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle',
'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'flags', 'float_info', 'getcheckinterval', 'getdefaultencoding',
...more names omitted...]
```

Only some of the many names are displayed here; run these statements on your machine to see the full list.

To find out what attributes are provided in built-in object types, run `dir` on a literal (or existing instance) of the desired type. For example, to see list and string attributes, you can pass empty objects:

```
>>> dir([])
['__add__', '__class__', '__contains__', ...more...
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']

>>> dir('')
['__add__', '__class__', '__contains__', ...more...
'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', '
maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
...more names omitted...]
```

`dir` results for any built-in type include a set of attributes that are related to the implementation of that type (technically, operator overloading methods); they all begin and end with double underscores to make them distinct, and you can safely ignore them at this point in the book.

Incidentally, you can achieve the same effect by passing a type name to `dir` instead of a literal:

```
>>> dir(str) == dir('')          # Same result as prior example
True
>>> dir(list) == dir([])
True
```

This works because names like `str` and `list` that were once type converter functions are actually names of types in Python today; calling one of these invokes its constructor to generate an instance of that type. I'll have more to say about constructors and operator overloading methods when we discuss classes in Part VI.

The `dir` function serves as a sort of memory-jogger—it provides a list of attribute names, but it does not tell you anything about what those names mean. For such extra information, we need to move on to the next documentation source.

Docstrings: __doc__

Besides # comments, Python supports documentation that is automatically attached to objects and retained at runtime for inspection. Syntactically, such comments are coded as strings at the tops of module files and function and class statements, before any other executable code (# comments are OK before them). Python automatically stuffs the strings, known as *docstrings*, into the `__doc__` attributes of the corresponding objects.

User-defined docstrings

For example, consider the following file, *docstrings.py*. Its docstrings appear at the beginning of the file and at the start of a function and a class within it. Here, I've used triple-quoted block strings for multiline comments in the file and the function, but any sort of string will work. We haven't studied the `def` or `class` statements in detail yet, so ignore everything about them except the strings at their tops:

```
"""
Module documentation
Words Go Here
"""

spam = 40

def square(x):
    """
    function documentation
    can we have your liver then?
    """
    return x ** 2          # square

class Employee:
    "class documentation"
    pass

print(square(4))
print(square.__doc__)
```

The whole point of this documentation protocol is that your comments are retained for inspection in `__doc__` attributes after the file is imported. Thus, to display the docstrings associated with the module and its objects, we simply import the file and print their `__doc__` attributes, where Python has saved the text:

```
>>> import docstrings
16

    function documentation
    can we have your liver then?

>>> print(docstrings.__doc__)

Module documentation
Words Go Here

>>> print(docstrings.square.__doc__)

    function documentation
    can we have your liver then?

>>> print(docstrings.Employee.__doc__)
    class documentation
```

Note that you will generally want to use `print` to print docstrings; otherwise, you'll get a single string with embedded newline characters.

You can also attach docstrings to *methods* of classes (covered in Part VI), but because these are just `def` statements nested in `class` statements, they're not a special case. To fetch the docstring of a method function inside a class within a module, you would simply extend the path to go through the class: `module.class.method.__doc__` (we'll see an example of method docstrings in Chapter 28).

Docstring standards

There is no broad standard about what should go into the text of a docstring (although some companies have internal standards). There have been various markup language and template proposals (e.g., HTML or XML), but they don't seem to have caught on in the Python world. And frankly, convincing Python programmers to document their code using handcoded HTML is probably not going to happen in our lifetimes!

Documentation tends to have a low priority amongst programmers in general. Usually, if you get any comments in a file at all, you count yourself lucky. I strongly encourage you to document your code liberally, though—it really is an important part of well-written programs. The point here is that there is presently no standard on the structure of docstrings; if you want to use them, anything goes today.

Built-in docstrings

As it turns out, built-in modules and objects in Python use similar techniques to attach documentation above and beyond the attribute lists returned by `dir`. For example, to see an actual human-readable description of a built-in module, import it and print its `__doc__` string:

```
>>> import sys
>>> print(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
...more text omitted...
```

Functions, classes, and methods within built-in modules have attached descriptions in their `__doc__` attributes as well:

```
>>> print(sys.getrefcount.__doc__)
getrefcount(object) -> integer

Return the reference count of object.  The count returned is generally
one higher than you might expect, because it includes the (temporary)
...more text omitted...
```

You can also read about built-in functions via their docstrings:

```
>>> print(int.__doc__)
int(x[, base]) -> integer

Convert a string or number to an integer, if possible.  A floating
point argument will be truncated towards zero (this does not include a
...more text omitted...

>>> print(map.__doc__)
map(func, *iterables) --> map object

Make an iterator that computes the function using arguments from
each of the iterables.  Stops when the shortest iterable is exhausted.
```

You can get a wealth of information about built-in tools by inspecting their docstrings this way, but you don't have to—the `help` function, the topic of the next section, does this automatically for you.

PyDoc: The help Function

The docstring technique proved to be so useful that Python now ships with a tool that makes docstrings even easier to display. The standard *PyDoc* tool is Python code that knows how to extract docstrings and associated structural information and format them into nicely arranged reports of various types. Additional tools for extracting and formatting docstrings are available in the open source domain (including tools that may support structured text—search the Web for pointers), but Python ships with PyDoc in its standard library.

There are a variety of ways to launch PyDoc, including command-line script options (see the Python library manual for details). Perhaps the two most prominent PyDoc interfaces are the built-in `help` function and the PyDoc GUI/HTML interface. The `help` function invokes PyDoc to generate a simple textual report (which looks much like a "manpage" on Unix-like systems):

```
>>> import sys
>>> help(sys.getrefcount)
Help on built-in function getrefcount in module sys:

getrefcount(...)
    getrefcount(object) -> integer

    Return the reference count of object.  The count returned is generally
    one higher than you might expect, because it includes the (temporary)
    ...more omitted...
```

Note that you do not have to import `sys` in order to call `help`, but you do have to import `sys` to get help on `sys`; it expects an object reference to be passed in. For larger objects such as modules and classes, the `help` display is broken down into multiple sections, a few of which are shown here. Run this interactively to see the full report:

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://docs.python.org/library/sys

DESCRIPTION
    This module provides access to some objects used or maintained by the
    interpreter and to functions that interact strongly with the interpreter.
    ...more omitted...

FUNCTIONS
    __displayhook__ = displayhook(...)
        displayhook(object) -> None

        Print an object to sys.stdout and also save it in builtins.
        ...more omitted...

DATA
    __stderr__ = <io.TextIOWrapper object at 0x0236E950>
    __stdin__ = <io.TextIOWrapper object at 0x02366550>
    __stdout__ = <io.TextIOWrapper object at 0x02366E30>
    ...more omitted...
```

Some of the information in this report is docstrings, and some of it (e.g., function call patterns) is structural information that PyDoc gleans automatically by inspecting objects' internals, when available. You can also use `help` on built-in functions, methods, and types. To get help for a built-in type, use the type name (e.g., `dict` for dictionary, `str` for string, `list` for list). You'll get a large display that describes all the methods available for that type:

```
>>> help(dict)
Help on class dict in module builtins:

class dict(object)
 |  dict() -> new empty dictionary.
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 ...more omitted...

>>> help(str.replace)
Help on method_descriptor:

replace(...)
    S.replace (old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    ...more omitted...

>>> help(ord)
```

```
Help on built-in function ord in module builtins:

ord(...)
    ord(c) -> integer

    Return the integer ordinal of a one-character string.
```

Finally, the `help` function works just as well on your modules as it does on built-ins. Here it is reporting on the *docstrings.py* file we coded earlier. Again, some of this is docstrings, and some is information automatically extracted by inspecting objects' structures:

```
>>> import docstrings
>>> help(docstrings.square)
Help on function square in module docstrings:

square(x)
    function documentation
    can we have your liver then?

>>> help(docstrings.Employee)
Help on class Employee in module docstrings:

class Employee(builtins.object)
 |  class documentation
 |
 |  Data descriptors defined here:
 ...more omitted...

>>> help(docstrings)
Help on module docstrings:

NAME
    docstrings

FILE
    c:\misc\docstrings.py

DESCRIPTION
    Module documentation
    Words Go Here

CLASSES
    builtins.object
        Employee

    class Employee(builtins.object)
     |  class documentation
     |
     |  Data descriptors defined here:
     ...more omitted...

FUNCTIONS
    square(x)
        function documentation
```

```
            can we have your liver then?

    DATA
        spam = 40
```

PyDoc: HTML Reports

The `help` function is nice for grabbing documentation when working interactively. For a more grandiose display, however, PyDoc also provides a GUI interface (a simple but portable Python/tkinter script) and can render its report in HTML page format, viewable in any web browser. In this mode, PyDoc can run locally or as a remote server in client/server mode; reports contain automatically created hyperlinks that allow you to click your way through the documentation of related components in your application.

To start PyDoc in this mode, you generally first launch the search engine GUI captured in Figure 15-1. You can start this either by selecting the "Module Docs" item in Python's Start button menu on Windows, or by launching the *pydoc.py* script in Python's standard library directory: *Lib* on Windows (run `pydoc.py` with a `-g` command-line argument). Enter the name of a module you're interested in, and press the Enter key; PyDoc will march down your module import search path (`sys.path`) looking for references to the requested module.

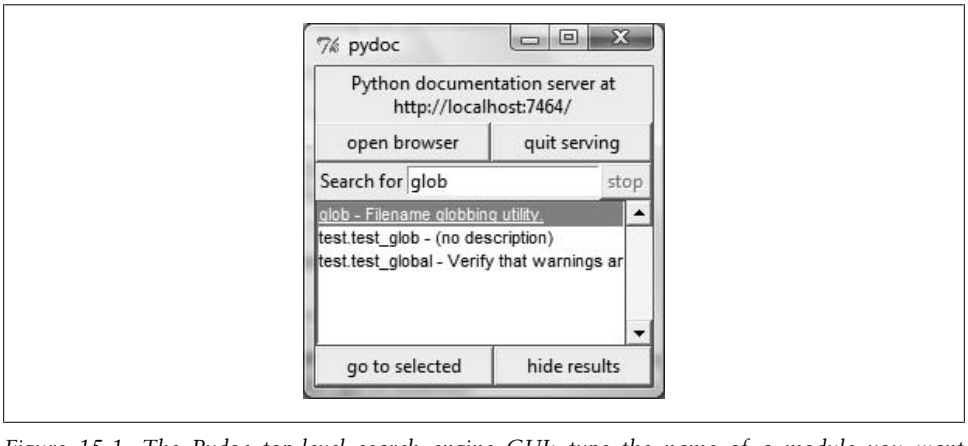

Figure 15-1. The Pydoc top-level search engine GUI: type the name of a module you want documentation for, press Enter, select the module, and then press "go to selected" (or omit the module name and press "open browser" to see all available modules).

Once you've found a promising entry, select it and click "go to selected." PyDoc will spawn a web browser on your machine to display the report rendered in HTML format. Figure 15-2 shows the information PyDoc displays for the built-in `glob` module.

Notice the hyperlinks in the Modules section of this page—you can click these to jump to the PyDoc pages for related (imported) modules. For larger pages, PyDoc also generates hyperlinks to sections within the page.

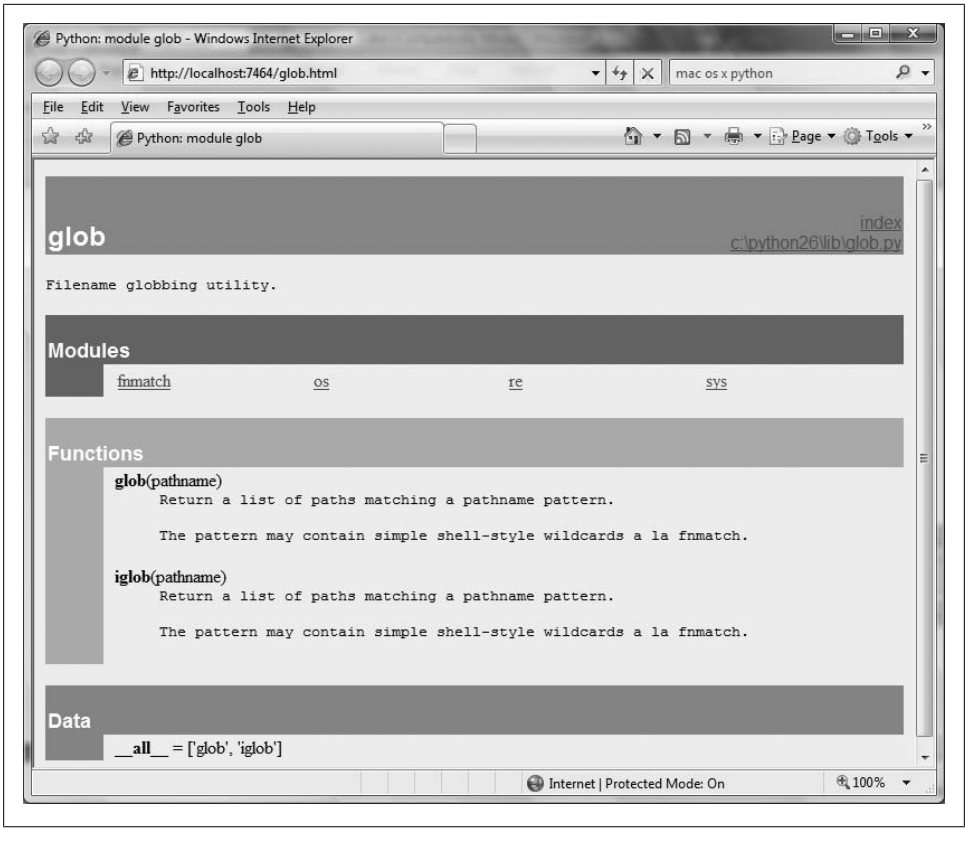

Figure 15-2. When you find a module in the Figure 15-1 GUI (such as this built-in standard library module) and press "go to selected," the module's documentation is rendered in HTML and displayed in a web browser window like this one.

Like the `help` function interface, the GUI interface works on user-defined modules as well as built-ins. Figure 15-3 shows the page generated for our *docstrings.py* module file.

PyDoc can be customized and launched in various ways we won't cover here; see its entry in Python's standard library manual for more details. The main thing to take away from this section is that PyDoc essentially gives you implementation reports "for free"—if you are good about using docstrings in your files, PyDoc does all the work of collecting and formatting them for display. PyDoc only helps for objects like functions and modules, but it provides an easy way to access a middle level of documentation for such tools—its reports are more useful than raw attribute lists, and less exhaustive than the standard manuals.

Cool PyDoc trick of the day: If you leave the module name empty in the top input field of the window in Figure 15-1 and press the "open browser" button, PyDoc will produce a web page containing a hyperlink to every module you can possibly import on your computer. This includes Python standard library modules, modules of third-party

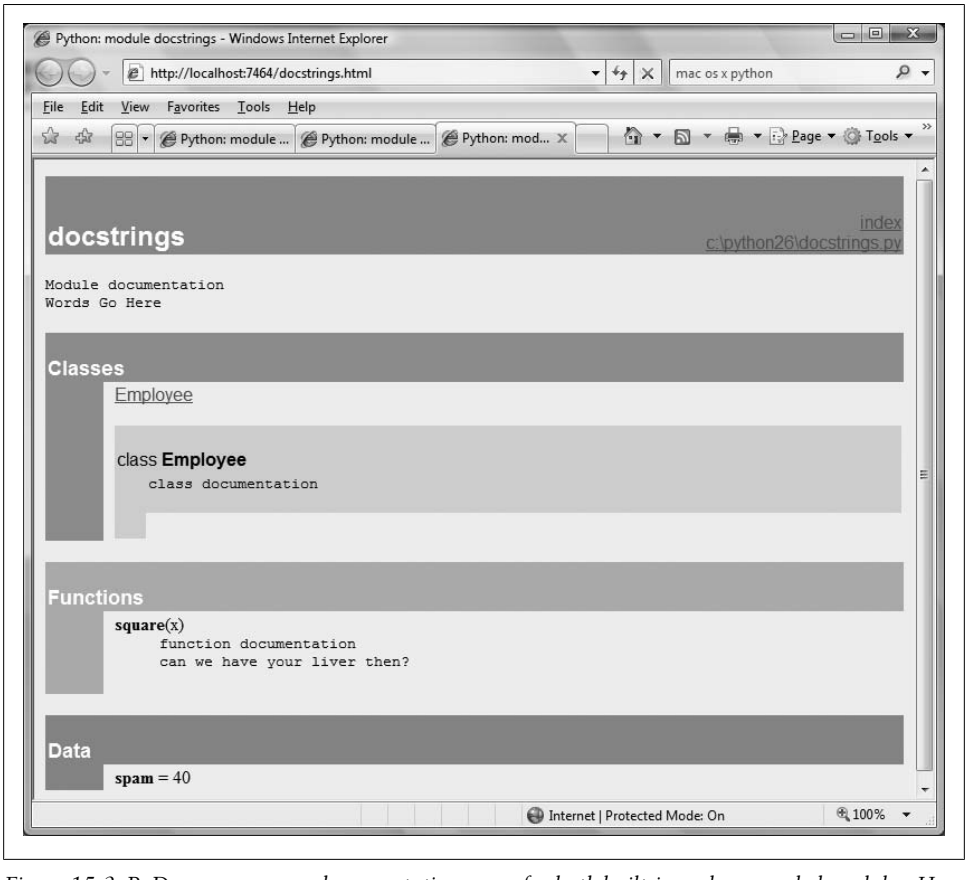

Figure 15-3. PyDoc can serve up documentation pages for both built-in and user-coded modules. Here is the page for a user-defined module, showing all its documentation strings (docstrings) extracted from the source file.

extensions you may have installed, user-defined modules on your import search path, and even statically or dynamically linked-in C-coded modules. Such information is hard to come by otherwise without writing code that inspects a set of module sources.

PyDoc can also be run to save the HTML documentation for a module in a file for later viewing or printing; see its documentation for pointers. Also, note that PyDoc might not work well if run on scripts that read from standard input—PyDoc imports the target module to inspect its contents, and there may be no connection for standard input text when it is run in GUI mode. Modules that can be imported without immediate input requirements will always work under PyDoc, though.

The Standard Manual Set

For the complete and most up-to-date description of the language and its toolset, Python's standard manuals stand ready to serve. Python's manuals ship in HTML and other formats, and they are installed with the Python system on Windows—they are available in your Start button's menu for Python, and they can also be opened from the Help menu within IDLE. You can also fetch the manual set separately from *http://www.python.org* in a variety of formats, or read them online at that site (follow the Documentation link). On Windows, the manuals are a compiled help file to support searches, and the online versions at the Python website include a web-based search page.

When opened, the Windows format of the manuals displays a root page like that in Figure 15-4. The two most important entries here are most likely the Library Reference (which documents built-in types, functions, exceptions, and standard library modules) and the Language Reference (which provides a formal description of language-level details). The tutorial listed on this page also provides a brief introduction for newcomers.

Figure 15-4. Python's standard manual set, available online at http://www.python.org, from IDLE's Help menu, and in the Windows Start button menu. It's a searchable help file on Windows, and there is a search engine for the online version. Of these, the Library Reference is the one you'll want to use most of the time.

Web Resources

At the official Python website (*http://www.python.org*), you'll find links to various Python resources, some of which cover special topics or domains. Click the Documentation link to access an online tutorial and the Beginners Guide to Python. The site also lists non-English Python resources.

You will find numerous Python wikis, blogs, websites, and a host of other resources on the Web today. To sample the online community, try searching for a term like "Python programming" in Google.

Published Books

As a final resource, you can choose from a large collection of reference books for Python. Bear in mind that books tend to lag behind the cutting edge of Python changes, partly because of the work involved in writing, and partly because of the natural delays built into the publishing cycle. Usually, by the time a book comes out, it's three or more months behind the current Python state. Unlike standard manuals, books are also generally not free.

Still, for many, the convenience and quality of a professionally published text is worth the cost. Moreover, Python changes so slowly that books are usually still relevant years after they are published, especially if their authors post updates on the Web. See the Preface for pointers to other Python books.

Common Coding Gotchas

Before the programming exercises for this part of the book, let's run through some of the most common mistakes beginners make when coding Python statements and programs. Many of these are warnings I've thrown out earlier in this part of the book, collected here for ease of reference. You'll learn to avoid these pitfalls once you've gained a bit of Python coding experience, but a few words now might help you avoid falling into some of these traps initially:

- **Don't forget the colons**. Always remember to type a `:` at the end of compound statement headers (the first line of an `if`, `while`, `for`, etc.). You'll probably forget at first (I did, and so have most of my 3,000 Python students over the years), but you can take some comfort from the fact that it will soon become an unconscious habit.
- **Start in column 1**. Be sure to start top-level (unnested) code in column 1. That includes unnested code typed into module files, as well as unnested code typed at the interactive prompt.

- **Blank lines matter at the interactive prompt**. Blank lines in compound statements are always ignored in module files, but when you're typing code at the interactive prompt, they end the statement. In other words, blank lines tell the interactive command line that you've finished a compound statement; if you want to continue, don't hit the Enter key at the `...` prompt (or in IDLE) until you're really done.
- **Indent consistently**. Avoid mixing tabs and spaces in the indentation of a block, unless you know what your text editor does with tabs. Otherwise, what you see in your editor may not be what Python sees when it counts tabs as a number of spaces. This is true in any block-structured language, not just Python—if the next programmer has her tabs set differently, she will not understand the structure of your code. It's safer to use all tabs or all spaces for each block.
- **Don't code C in Python**. A reminder for C/C++ programmers: you don't need to type parentheses around tests in `if` and `while` headers (e.g., `if (X==1):`). You can, if you like (any expression can be enclosed in parentheses), but they are fully superfluous in this context. Also, do not terminate all your statements with semicolons; it's technically legal to do this in Python as well, but it's totally useless unless you're placing more than one statement on a single line (the end of a line normally terminates a statement). And remember, don't embed assignment statements in `while` loop tests, and don't use `{}` around blocks (indent your nested code blocks consistently instead).
- **Use simple `for` loops instead of `while` or `range`**. Another reminder: a simple `for` loop (e.g., `for x in seq:`) is almost always simpler to code and quicker to run than a `while`- or `range`-based counter loop. Because Python handles indexing internally for a simple `for`, it can sometimes be twice as fast as the equivalent `while`. Avoid the temptation to count things in Python!
- **Beware of mutables in assignments**. I mentioned this in Chapter 11: you need to be careful about using mutables in a multiple-target assignment (`a = b = []`), as well as in an augmented assignment (`a += [1, 2]`). In both cases, in-place changes may impact other variables. See Chapter 11 for details.
- **Don't expect results from functions that change objects in-place**. We encountered this one earlier, too: in-place change operations like the `list.append` and `list.sort` methods introduced in Chapter 8 do not return values (other than `None`), so you should call them without assigning the result. It's not uncommon for beginners to say something like `mylist = mylist.append(X)` to try to get the result of an `append`, but what this actually does is assign `mylist` to `None`, not to the modified list (in fact, you'll lose your reference to the list altogether).

  A more devious example of this pops up in Python 2.X code when trying to step through dictionary items in a sorted fashion. It's fairly common to see code like `for k in D.keys().sort():`. This almost works—the `keys` method builds a keys list, and the `sort` method orders it—but because the `sort` method returns `None`, the loop fails because it is ultimately a loop over `None` (a nonsequence). This fails even

sooner in Python 3.0, because dictionary keys are views, not lists! To code this correctly, either use the newer `sorted` built-in function, which returns the sorted list, or split the method calls out to statements: `Ks = list(D.keys())`, then `Ks.sort()`, and finally, `for k in Ks:`. This, by the way, is one case where you'll still want to call the `keys` method explicitly for looping, instead of relying on the dictionary iterators—iterators do not sort.

- **Always use parentheses to call a function**. You must add parentheses after a function name to call it, whether it takes arguments or not (e.g., use `function()`, not `function`). In Part IV, we'll see that functions are simply objects that have a special operation—a call that you trigger with the parentheses.

  In classes, this problem seems to occur most often with files; it's common to see beginners type `file.close` to close a file, rather than `file.close()`. Because it's legal to reference a function without calling it, the first version with no parentheses succeeds silently, but it does not close the file!

- **Don't use extensions or paths in imports and reloads**. Omit directory paths and file suffixes in `import` statements (e.g., say `import mod`, not `import mod.py`). (We discussed module basics in Chapter 3 and will continue studying modules in Part V.) Because modules may have other suffixes besides *.py* (*.pyc*, for instance), hardcoding a particular suffix is not only illegal syntax, but doesn't make sense. Any platform-specific directory path syntax comes from module search path settings, not the `import` statement.

Chapter Summary

This chapter took us on a tour of program documentation—both documentation we write ourselves for our own programs, and documentation available for built-in tools. We met docstrings, explored the online and manual resources for Python reference, and learned how PyDoc's `help` function and web page interface provide extra sources of documentation. Because this is the last chapter in this part of the book, we also reviewed common coding mistakes to help you avoid them.

In the next part of this book, we'll start applying what we already know to larger program constructs: functions. Before moving on, however, be sure to work through the set of lab exercises for this part of the book that appear at the end of this chapter. And even before that, let's run through this chapter's quiz.

Test Your Knowledge: Quiz

1. When should you use documentation strings instead of hash-mark comments?
2. Name three ways you can view documentation strings.

3. How can you obtain a list of the available attributes in an object?
4. How can you get a list of all available modules on your computer?
5. Which Python book should you purchase after this one?

Test Your Knowledge: Answers

1. Documentation strings (docstrings) are considered best for larger, functional documentation, describing the use of modules, functions, classes, and methods in your code. Hash-mark comments are today best limited to micro-documentation about arcane expressions or statements. This is partly because docstrings are easier to find in a source file, but also because they can be extracted and displayed by the PyDoc system.
2. You can see docstrings by printing an object's `__doc__` attribute, by passing it to PyDoc's `help` function, and by selecting modules in PyDoc's GUI search engine in client/server mode. Additionally, PyDoc can be run to save a module's documentation in an HTML file for later viewing or printing.
3. The built-in `dir(X)` function returns a list of all the attributes attached to any object.
4. Run the PyDoc GUI interface, leave the module name blank, and select "open browser"; this opens a web page containing a link to every module available to your programs.
5. Mine, of course. (Seriously, the Preface lists a few recommended follow-up books, both for reference and for application tutorials.)

Test Your Knowledge: Part III Exercises

Now that you know how to code basic program logic, the following exercises will ask you to implement some simple tasks with statements. Most of the work is in exercise 4, which lets you explore coding alternatives. There are always many ways to arrange statements, and part of learning Python is learning which arrangements work better than others.

See Part III in Appendix B for the solutions.

1. *Coding basic loops*.
   a. Write a `for` loop that prints the ASCII code of each character in a string named `S`. Use the built-in function `ord(character)` to convert each character to an ASCII integer. (Test it interactively to see how it works.)
   b. Next, change your loop to compute the sum of the ASCII codes of all the characters in a string.

c. Finally, modify your code again to return a new list that contains the ASCII codes of each character in the string. Does the expression `map(ord, S)` have a similar effect? (Hint: see Chapter 14.)

2. *Backslash characters*. What happens on your machine when you type the following code interactively?

```
for i in range(50):
    print('hello %d\n\a' % i)
```

Beware that if it's run outside of the IDLE interface this example may beep at you, so you may not want to run it in a crowded lab. IDLE prints odd characters instead of beeping (see the backslash escape characters in Table 7-2).

3. *Sorting dictionaries*. In Chapter 8, we saw that dictionaries are unordered collections. Write a `for` loop that prints a dictionary's items in sorted (ascending) order. (Hint: use the dictionary `keys` and list `sort` methods, or the newer `sorted` built-in function.)

4. *Program logic alternatives*. Consider the following code, which uses a `while` loop and `found` flag to search a list of powers of 2 for the value of 2 raised to the fifth power (32). It's stored in a module file called *power.py*.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = False
i = 0
while not found and i < len(L):
   if 2 ** X == L[i]:
       found = True
   else:
       i = i+1

if found:
    print('at index', i)
else:
    print(X, 'not found')

C:\book\tests> python power.py
at index 5
```

As is, the example doesn't follow normal Python coding techniques. Follow the steps outlined here to improve it (for all the transformations, you may either type your code interactively or store it in a script file run from the system command line—using a file makes this exercise much easier):

a. First, rewrite this code with a `while` loop `else` clause to eliminate the `found` flag and final `if` statement.

b. Next, rewrite the example to use a `for` loop with an `else` clause, to eliminate the explicit list-indexing logic. (Hint: to get the index of an item, use the list `index` method—`L.index(X)` returns the offset of the first `X` in list `L`.)

c. Next, remove the loop completely by rewriting the example with a simple `in` operator membership expression. (See Chapter 8 for more details, or type this to test: `2 in [1,2,3]`.)

d. Finally, use a `for` loop and the list `append` method to generate the powers-of-2 list (`L`) instead of hardcoding a list literal.

Deeper thoughts:

e. Do you think it would improve performance to move the `2 ** X` expression outside the loops? How would you code that?

f. As we saw in exercise 1, Python includes a `map(function, list)` tool that can generate a powers-of-2 list, too: `map(lambda x: 2 ** x, range(7))`. Try typing this code interactively; we'll meet `lambda` more formally in Chapter 19.