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, cover the “ternary” `if` expression, and fill in some details on truth tests in general.

if Statements

In simple terms, the Python `if` statement selects actions to perform. Along with its expression counterpart, 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 such as selections and loops).

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 you're 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 (as we'll see later, the word `True` is its equivalent), 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("shave and a haircut")
... 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 block nested under the `else`. 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, you usually code *multiway branching* as a series of `if`/`elif` tests, as in the prior example, and occasionally by indexing dictionaries or searching lists. Because dictionaries and lists can be built at runtime dynamically, they are sometimes more flexible than hardcoded `if` logic in your script:

```
>>> 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 the following:

```
>>> if choice == 'spam':           # The equivalent if statement
...     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
```

Though it's perhaps more readable, the potential downside of an `if` like this is that, short of constructing it as a string and running it with tools like the prior chapter's `eval` or `exec`, you cannot construct it at runtime as easily as a dictionary. In more dynamic programs, data structures offer added flexibility.

Handling switch defaults

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 with the `try` statement introduced in the preceding chapter. All of the same techniques can be used here to code a default action in a dictionary-based multiway branch. As a review in the context of this use case, 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
```

And the try statement is a general way to handle defaults by catching and handling the exceptions they'd otherwise trigger (for more on exceptions, see Chapter 11's overview and Part VII's full treatment):

```
>>> try:
...     print(branch[choice])
... except KeyError:
...     print('Bad choice')
...
Bad choice
```

Handling larger actions

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, they may be coded as function names or inline lambdas, and they are called by adding parentheses to trigger their actions. Here's an abstract sampler, but stay tuned for a rehash of this topic in Chapter 19 after we've learned more about function definition:

```
def function(): ...
def default(): ...

branch = {'spam': lambda: ...,                # A table of callable function objects
          'ham':  function,
          'eggs': lambda: ...}

branch.get(choice, default)()
```

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 Revisited

I introduced Python's syntax model in Chapter 10. Now that we're stepping up to larger statements like `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 as a *sequence*, but statements like `if` (as well as loops and exceptions) 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. As a special case, statements can span lines and be combined on a line with special syntax.
- **Compound statements = header + ":" + indented statements**. All Python *compound statements*—those with nested statements—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. As a special case, blocks can show up on the same line as the header if they are simple noncompound code.
- **Blank lines, spaces, and comments are usually ignored**. Blank lines are both optional and 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 like PyDoc. Docstrings are part of Python's larger documentation strategy and are covered in the last chapter in this part of the book.

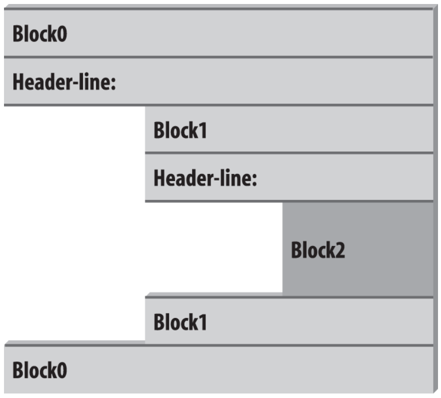

Figure 12-1. Nested blocks of code: a nested block starts with a statement indented further to the right and ends with either a statement that is indented less, or the end of the file.

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

As introduced in Chapter 10, 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. Compound statement bodies can appear on the header's line in some cases we'll explore later, but most are indented under it.

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')
```

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)                    # Prints 8 "SPAM"
    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 simpler in practice than its details might initially 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.X

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 completely apart from Python's syntax rules—tabs may look very different in the next programmer's editor than they do in yours.

In fact, Python 3.X 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.X 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.X's error case is equivalent to 2.X'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.X and 2.7.
- **Statements may span multiple lines if they end in a backslash**. This is a somewhat outdated feature that's not generally recommended, 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 rarely used today. This approach is also error-prone: accidentally forgetting a `\` usually generates a syntax error and might even cause the next line to be silently mistaken (i.e., without warning) for 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 it's 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 it is 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 just described. 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.X and 2.7, 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, and are obvious
```

In fact, backslashes are generally frowned on by most Python developers, 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:[1]

```
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'`; as we also saw in Chapter 7, `#` 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 contains just simple (noncompound) statements. You'll most often see this used for simple `if` statements with a single test and action, as in the interactive loops we coded in Chapter 10:

```
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.

Truth Values and Boolean 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

1. Candidly, it was a bit surprising that backslash continuations were not removed in Python 3.0, given the broad scope of its other changes! See the 3.0 changes tables in Appendix C 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. You should generally avoid backslash continuations in new Python code, even if you developed the habit in your C programming days.

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:

- All objects have an inherent Boolean true or false value.
- 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.
- Boolean operators stop evaluating ("short circuit") as soon as a result is known.

The `if` statement takes action on truth values, but Boolean operators are used to combine the results of other tests in richer ways to produce new truth values. More formally, 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 Chapter 5 and Chapter 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 as soon as the result is known:

```
>>> 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—it determines the result because true or anything is always true. 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.

Python 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 if the left operand is a *false* object because it determines the result—false and anything is always false:

```
>>> 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 according to the normal definitions of or and and. 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 384 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 more mnemonic replacement in 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, running just `Y` or `Z` but not both. Here are some examples of it in action:

```
>>> A = 't' if 'spam' else 'f'       # For strings, nonempty means 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 as the preceding section described:

```
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 `X` if false the `and` skips `Y`, and the `or` simply returns `Z`. In other words, we get "if `X` then `Y` else `Z`." This is equivalent to the ternary form:

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

The `and`/`or` combination form 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 `if/else` expression when you need this structure, 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 as offsets to pick true and false values from a list:

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

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 Python code written by ex–C programmers who haven't quite let go of their dark coding pasts).[2]

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
```

assigns `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 the short-circuit evaluation of Boolean operators and the `if/else`, because it may prevent actions from running. Expressions on the right of a Boolean operator, for example, 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` 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 Boolean values `True` and `False` are the same as the integers `1` and `0` and are useful for initializing variables

2. In fact, Python's `Y if X else Z` has a slightly different order than C's `X ? Y : Z`, and uses more readable words. Its differing order was reportedly chosen in response to analysis of common usage patterns in Python code. According to the Python folklore, 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. If you have to work at packing logic into expressions like this, statements are probably your better bet.

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

Also watch for related discussion in 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 2.7). 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.

Finally, and as a preview, other tools in Python have roles similar to the `or` chains at the start of this sidebar: the `filter` call and list comprehensions we'll meet later can be used to select true values when the set of candidates isn't known until runtime (though they evaluate all values and return all that are true), and the `any` and `all` built-ins can be used to test if any or all items in a collection are true (though they don't select an item):

```
>>> L = [1, 0, 2, 0, 'spam', '', 'ham', []]
>>> list(filter(bool, L))                    # Get true values
[1, 2, 'spam', 'ham']
>>> [x for x in L if x]                      # Comprehensions
[1, 2, 'spam', 'ham']
>>> any(L), all(L)                           # Aggregate truth
(True, False)
```

As seen in Chapter 9, the `bool` function here simply returns its argument's true or false value, as though it were tested in an `if`. Watch for more on these related tools in Chapter 14, Chapter 19, and Chapter 20.

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 values and tests in more depth than we were able to previously. Along the way, we also looked at how to code multiway branching in Python, learned about the `if/else` expression introduced in Python 2.5, and explored some common ways that Boolean values crop up in code.

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 or flexible. 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. Backslash continuations work too, but are broadly discouraged in the Python world.
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. In all these roles, they serve as a more mnemonic and hence readable alternative to `1` and `0`.

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 or other iterable 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 and the `while` statement is skipped.

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 normally 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
    statements                      # Loop body
else:                               # Optional else
    statements                      # 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*—it's not really immortal, but you may need a Ctrl-C key combination to forcibly terminate one:

```
>>> 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 through the items in a string more easily with a `for` loop.

```
>>> x = 'spam'
>>> while x:                      # While x is not empty
...     print(x, end=' ')         # In 2.X use print x,
...     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. This may leave your input prompt in an odd state at the end of your output; type Enter to reset. Python 2.X readers: also remember to use a trailing comma instead of `end` in the `print`s like this.

The following code counts from the value of `a` up to, but not including, `b`. We'll also 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, so that the loop's body is always run at least once:

```
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, `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 test:
    statements
    if test: break                  # Exit loop now, skip else if present
    if test: continue               # Go to top of loop now, to test1
else:
    statements                      # 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.X (but not 2.X) 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.X—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 (modulus) 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 here inside an `if` test; the `print` is only reached if the `continue` is not run. If this sounds similar to a "go to" in other languages, it should. Python has no "go to" statement, but because `continue` lets you jump about in a program, many of the warnings about readability and maintainability you may have heard about "go to" 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=' ')
```

Later in this book, we'll also learn that raised and caught exceptions can also emulate "go to" statements in limited and structured ways; stay tuned for more on this technique in Chapter 36 where we will learn how to use it to break out of multiple nested loops, a feat not possible with the next section's topic alone.

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.X) and exits when the user enters “stop” for the name request:

```
>>> while True:
...     name = input('Enter name:')            # Use raw_input() in 2.X
...     if name == 'stop': break
...     age  = input('Enter age: ')
...     print('Hello', name, '=>', int(age) ** 2)
...
Enter name:bob
Enter age: 40
Hello bob => 1600
Enter name:sue
Enter age: 30
Hello sue => 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 36, you’ll see that `input` also raises an exception at end-of-file (e.g., if the user types Ctrl-Z on Windows or Ctrl-D on Unix); 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. Trace through this code to see how this works.

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.X 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 (and go unused by some veterans; I've met some who didn't even know there *was* an `else` on loops!). In general terms, the loop `else` simply 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 (this code is intentionally abstract and incomplete; `x` is a sequence and `match` is a tester function to be defined):

```
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 is, each statement must generally appear on a line by itself, not nested in a larger construct. That means this common C language coding pattern won't work in Python:

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

or move the assignment into the loop with tests:

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

or move the first assignment outside the loop:

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

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 such C loops as well and be more Pythonic, but C doesn't have a directly analogous tool:

```
for x in obj: ...process x...
```

for Loops

The `for` loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable object. The `for` statement works on strings, lists, tuples, and other built-in iterables, as well as new user-defined objects that we'll learn how to create later with classes. We met `for` briefly in Chapter 4 and in conjunction with 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:                                 # Optional else part
    statements                        # If we didn't hit a 'break'
```

When Python runs a `for` loop, it assigns the items in the iterable `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 unique about this name; 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 often 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 learn 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`. 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)                        # 2.X: prints with enclosing tuple "()"
...
```

```
1 2
3 4
5 6
```

But 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
```

Even this is not a special case, though—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, simply 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.X extended sequence assignment in for loops

In fact, because the loop variable in a `for` loop can be any assignment target, we can also use Python 3.X'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.X, 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.X, 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.X
...     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`.

This example is illustrative, but it may be 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 is similar, but builds a list as it goes for later use instead of printing. It performs a typical data-structure task with a `for`—collecting common items in two sequences (strings)—and serves as a rough 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.

This code also exhibits the classic *list comprehension* pattern—collecting a results list with an iteration and optional filter test—and could be coded more concisely too:

```
>>> [x for x in seq1 if x in seq2]       # Let Python collect results
['s', 'a', 'm']
```

But you'll have to read on to the next chapter for the rest of this story.

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           # Empty string means end-of-file
    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.rstrip())         # 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.rstrip())

for line in open('test.txt'):    # Use iterators: best for text input
    print(line.rstrip())
```

Both of these versions work in both Python 2.X and 3.X. The first uses the file `readlines` method to load 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.

The last example is also generally the *best* option for text files—besides its simplicity, it works for arbitrarily large files because it doesn't load the entire file into memory all at once. The iterator version may also be the quickest, though I/O performance may vary per Python line and release.

File `readlines` calls can still be useful, though—to *reverse* a file's lines, for example, assuming its content can fit in memory. The `reversed` built-in accepts a sequence, but not an arbitrary iterable that generates values; in other words, a list works, but a file object doesn't:

```
for line in reversed(open('test.txt').readlines()): ...
```

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.X, because they are redundant. You should generally avoid them in new 2.X code too—use file iterators and `open` call in recent 2.X releases—but they may pop up in older code and resources.

See the library manual for more on the calls used here, and Chapter 14 for more on file line iterators. Also watch for the sidebar "Why You Will Care: Shell Commands and More" on page 411 in this chapter; it applies these same file tools to the `os.popen` command-line launcher to read program output. There's more on reading files in Chapter 37 too; as we'll see there, text and binary files have slightly different semantics in 3.X.

Loop Coding Techniques

The `for` loop we just studied subsumes most counter-style loops. It's generally simpler to code and often quicker to run than a `while`, so it's the first tool you should reach for whenever you need to step through a sequence or other iterable. In fact, as a general rule, you should *resist the temptation to count things in Python*—its iteration tools automate much of the work you do to loop over collections in lower-level languages like C.

Still, there are 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? What if you need indexes too?

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

- The built-in `range` function (available since Python 0.X) produces a series of successively higher integers, which can be used as indexes in a `for`.
- The built-in `zip` function (available since Python 2.0) returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a `for`.
- The built-in `enumerate` function (available since Python 2.3) generates both the values and indexes of items in an iterable, so we don't need to count manually.
- The built-in `map` function (available since Python 1.0) can have a similar effect to `zip` in Python 2.X, though this role is removed in 3.X.

Because `for` loops may run quicker than `while`-based counter loops, though, it's to your advantage to use tools like these that allow you to use `for` whenever possible. Let's look at each of these built-ins in turn, in the context of common use cases. As we'll see, their usage may differ slightly between 2.X and 3.X, and some of their applications are more valid than others.

Counter Loops: range

Our first loop-related function, `range`, is really a general tool that can be used in a variety of contexts. We met it briefly in Chapter 4. Although it's used most often to generate indexes in a `for`, you can use it anywhere you need a series of integers. In

Python 2.X range creates a physical *list*; in 3.X, `range` is an *iterable* that generates items on demand, so we need to wrap it in a `list` call to display its results all at once in 3.X only:

```
>>> 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]
```

We'll get more formal about iterables like this one in Chapter 14. There, we'll also see that Python 2.X has a cousin named `xrange`, which is like its `range` but doesn't build the result list in memory all at once. This is a space optimization, which is subsumed in 3.X by the generator behavior of its `range`.

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

Note that `for` loops force results from `range` automatically in 3.X, so we don't need to use a `list` wrapper here in 3.X (in 2.X we get a temporary list unless we call `xrange` instead).

Sequence Scans: while and range Versus for

The `range` call is also sometimes used to iterate over a sequence indirectly, though it's often not the best approach in this role. The easiest and generally 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 range/len iteration
...
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. If this seems like overkill, though, it's because it is: there's really no reason to work this hard in this example.

Although the `range`/`len` combination suffices in this role, it's probably not the best option. It may run slower, and it's also more work than we need to do. Unless you have a special indexing requirement, you're better off using the simple `for` loop form in Python:

```
>>> for item in X: print(item, end=' ')           # Use simple iteration if you can
```

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 almost always better. Like every good rule, though, there are plenty of exceptions—as the next section demonstrates.

Sequence Shufflers: range and len

Though not ideal for simple sequence scans, the coding pattern used in the prior example does allow us to do more specialized sorts of traversals when required. For example, some algorithms can make use of sequence reordering—to generate alternatives in searches, to test the effect of different value orderings, and so on. Such cases may require offsets in order to pull sequences apart and put them back together, as in the following; the range's integers provide a repeat count in the first, and a position for slicing in the second:

```
>>> S = 'spam'
>>> for i in range(len(S)):        # For repeat counts 0..3
...     S = S[1:] + S[:1]          # Move front item to end
...     print(S, end=' ')
...
pams amsp mspa spam

>>> S
'spam'
>>> for i in range(len(S)):        # For positions 0..3
...     X = S[i:] + S[:i]          # Rear part + front part
...     print(X, end=' ')
...
spam pams amsp mspa
```

Trace through these one iteration at a time if they seem confusing. The second creates the same results as the first, though in a different order, and doesn't change the original variable as it goes. Because both slice to obtain parts to concatenate, they also work on any type of sequence, and return sequences of the same type as that being shuffled—if you shuffle a list, you create reordered lists:

```
>>> L = [1, 2, 3]
>>> for i in range(len(L)):
...     X = L[i:] + L[:i]          # Works on any sequence type
...     print(X, end=' ')
...
[1, 2, 3] [2, 3, 1] [3, 1, 2]
```

We'll make use of code like this to test functions with different argument orderings in Chapter 18, and will extend it to functions, generators, and more complete permutations in Chapter 20—it's a widely useful tool.

Nonexhaustive Traversals: range Versus Slices

Cases like that of the prior section are valid applications for the `range`/`len` combination. We might also use this technique to 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.

In most cases, though, this is also probably not the "best practice" technique in Python today. If you really mean 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 potential advantage to using `range` here instead is space: slicing makes a copy of the string in both 2.X and 3.X, while `range` in 3.X and `xrange` in 2.X do not create a list; for very large strings, they may save memory.

Changing Lists: range Versus Comprehensions

Another common place where you may use the `range`/`len` combination with `for` 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 (maybe you're giving everyone a raise in an employee database 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                          # Changes x, not L
...
>>> 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; it's a piece of memory separate from the list.

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 might run more slowly depending on your Python (it does on 2.7 and 3.3, though less so on 3.3—we'll see how to verify this in Chapter 21):

```
>>> 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]
```

likely runs faster today and 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, and continue this story there.

Parallel Traversals: zip and map

Our next loop coding technique extends a loop's scope. 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*—not overlapping in time, but during the same loop. 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 (a list of names and addresses paired by position, perhaps):

```
>>> 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 a list in Python 2.X, but an iterable object in 3.X where we must wrap it in a `list` call to display all its results at once (again, there's more on iterables coming up in the next chapter):

```
>>> zip(L1, L2)
<zip object at 0x026523C8>
>>> list(zip(L1, L2))                    # list() required in 3.X, not 2.X
[(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))                # Three tuples for three arguments
[(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))                    # Truncates at len(shortest)
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

map equivalence in Python 2.X

In Python 2.X only, the related built-in `map` function pairs items from sequences in a similar fashion when passed `None` for its function argument, 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: pads to len(longest)
[('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.X. 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 Chapter 19 and Chapter 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.X and so must be passed to `list` to collect all its results at once in 3.X only):

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

This works the same as the following loop statement, but `map` is often quicker, as Chapter 21 will show:

```
>>> 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.X, because it largely overlaps with `zip` (and was, frankly, a bit at odds with `map`'s function-application purpose). In 3.X, either use `zip` or write loop code to pad results yourself. In fact, we'll see how to write such loop code in Chapter 20, after we've had a chance to study some additional iteration concepts.

Dictionary construction with zip

Let's look at another `zip` use case. Chapter 8 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`, let's explore more fully 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
{'eggs': 3, 'toast': 5, '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, collected from a user, parsed from a file, or obtained from another dynamic source:

```
>>> 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
{'eggs': 3, 'toast': 5, '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
{'eggs': 3, 'toast': 5, '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 32). 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 the related but richer concept, the list comprehension, which builds lists in a single expression; we'll also revisit Python 3.X and 2.7 dictionary comprehensions, an alternative to the `dict` call for zipped key/value pairs:

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

Generating Both Offsets and Items: enumerate

Our final loop helper function is designed to support dual usage modes. 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 all recent Python 2.X and 3.X releases (since 2.3) a new built-in named `enumerate` does the job for us—its net effect is to give loops a counter "for free," without sacrificing the simplicity of automatic iteration:

```
>>> 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 method called by the `next` built-in function, which returns an `(index, value)` tuple each time through the loop. The `for` steps through these tuples automatically, which allows us to unpack their values with tuple assignment, much as we did for `zip`:

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x0000000002A8B900>
>>> next(E)
(0, 's')
>>> next(E)
(1, 'p')
>>> next(E)
(2, 'a')
```

We don't normally see this machinery because all 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']

>>> for (i, l) in enumerate(open('test.txt')):
...     print('%s) %s' % (i, l.rstrip()))
...
0) aaaaaa
1) bbbbbb
2) cccccc
```

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

Why You Will Care: Shell Commands and More

An earlier sidebar showed loops applied to files. As briefly noted in Chapter 9, Python's related `os.popen` call also gives a file-like interface, for reading the outputs of spawned *shell commands*. Now that we've studied looping statements in full, here's an example of this tool in action—to run a shell command and read its standard output text, pass the command as a string to `os popen`, and read text from the file-like object it returns (if this triggers a Unicode encoding issue on your computer, Chapter 25's discussion of currency symbols may apply):

```
>>> import os
>>> F = os.popen('dir')                # Read line by line
>>> F.readline()
' Volume in drive C has no label.\n'
>>> F = os.popen('dir')                # Read by sized blocks
>>> F.read(50)
' Volume in drive C has no label.\n Volume Serial Nu'

>>> os.popen('dir').readlines()[0]     # Read all lines: index
' Volume in drive C has no label.\n'
>>> os.popen('dir').read()[:50]        # Read all at once: slice
' Volume in drive C has no label.\n Volume Serial Nu'

>>> for line in os.popen('dir'):       # File line iterator loop
...     print(line.rstrip())
...
 Volume in drive C has no label.
 Volume Serial Number is D093-D1F7
...and so on...
```

This runs a `dir` directory listing on Windows, but any program that can be started with a command line can be launched this way. We might use this scheme, for example, to display the output of the windows `systeminfo` command—`os.system` simply runs a shell command, but `os.popen` also connects to its streams; both of the following show the shell command's output in a simple console window, but the first might not in a GUI interface such as IDLE:

```
>>> os.system('systeminfo')
...output in console, popup in IDLE...
0
>>> for line in os.popen('systeminfo'): print(line.rstrip())

Host Name:                 MARK-VAIO
OS Name:                   Microsoft Windows 7 Professional
OS Version:                6.1.7601 Service Pack 1 Build 7601
...lots of system information text...
```

And once we have a command's output in text form, any string processing tool or technique applies—including display formatting and content parsing:

```
# Formatted, limited display
>>> for (i, line) in enumerate(os.popen('systeminfo')):
...     if i == 4: break
...     print('%05d) %s' % (i, line.rstrip()))
...
00000)
00001) Host Name:                 MARK-VAIO
00002) OS Name:                   Microsoft Windows 7 Professional
00003) OS Version:                6.1.7601 Service Pack 1 Build 7601

# Parse for specific lines, case neutral
>>> for line in os.popen('systeminfo'):
...    parts = line.split(':')
...    if parts and parts[0].lower() == 'system type':
...        print(parts[1].strip())
```

```
...
x64-based PC
```

We'll see `os.popen` in action again in Chapter 21, where we'll deploy it to read the results of a constructed command line that times code alternatives, and in Chapter 25, where it will be used to compare outputs of scripts being tested.

Tools like `os.popen` and `os.system` (and the `subprocess` module not shown here) allow you to leverage every command-line program on your computer, but you can also write emulators with in-process code. For example, simulating the Unix `awk` utility's ability to strip columns out of text files is almost trivial in Python, and can become a reusable function in the process:

```
# awk emulation: extract column 7 from whitespace-delimited file
for val in [line.split()[6] for line in open('input.txt')]:
    print(val)

# Same, but more explicit code that retains result
col7 = []
for line in open('input.txt'):
    cols = line.split()
    col7.append(cols[6])
for item in col7:  print(item)

# Same, but a reusable function (see next part of book)
def awker(file, col):
    return [line.rstrip().split()[col-1] for line in open(file)]

print(awker('input.txt', 7))              # List of strings
print(','.join(awker('input.txt', 7)))    # Put commas between
```

By itself, though, Python provides file-like access to a wide variety of data—including the text returned by *websites* and their pages identified by URL, though we'll have to defer to Part V for more on the package import used here, and other resources for more on such tools in general (e.g., this works in 2.X, but uses `urllib` instead of `urllib.request`, and returns text strings):

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://home.rmi.net/~lutz'):
...     print(line)
...
b'<HTML>\n'
b'\n'
b'<HEAD>\n'
b"<TITLE>Mark Lutz's Book Support Site</TITLE>\n"
...etc...
```

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 some of the details regarding their roles as iterables in Python 3.X were intentionally cut short.

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 give the rest of the picture behind the iterable tools we met here, such as `range` and `zip`, and study some of the subtleties of their operation. 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 or other 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

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*, which are a close cousin to the `for` loop that applies an expression to items in an iterable.

Because these tools are related to both the `for` loop and functions, we'll take a two-pass approach to covering them in this book, along with a postscript:

- This chapter introduces their basics in the context of looping tools, serving as something of a continuation of the prior chapter.
- Chapter 20 revisits them in the context of function-based tools, and extends the topic to include built-in and user-defined *generators*.
- Chapter 30 also provides a shorter final installment in this story, where we'll learn about user-defined iterable objects coded with *classes*.

In this chapter, we'll also sample additional iteration tools in Python, and touch on the new iterables available in Python 3.X—where the notion of iterables grows even more pervasive.

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.

Iterations: 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=' ')             # In 2.X: print x ** 2,
...
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.

Terminology in this topic tends to be a bit loose. The terms "iterable" and "iterator" are sometimes used interchangeably to refer to an object that supports iteration in general. For clarity, this book has a very strong preference for using the term *iterable* to refer to an object that supports the `iter` call, and *iterator* to refer to an object returned by an iterable on `iter` that supports the `next(I)` call. Both these calls are defined ahead.

That convention is not universal in either the Python world or this book, though; "iterator" is also sometimes used for tools that iterate. Chapter 20 extends this category with the term "generator"—which refers to objects that automatically support the iteration protocol, and hence are iterable—even though all iterables generate results!

The Iteration Protocol: File Iterators

One of the easiest ways to understand the iteration protocol is to see how it works with a built-in type such as the file. In this chapter, we'll be using the following input file to demonstrate:

```
>>> print(open('script2.py').read())
import sys
```

```
print(sys.path)
x = 2
print(x ** 32)

>>> open('script2.py').read()
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'
```

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('script2.py')       # Read a four-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()                 # Last lines may have a \n or not
'print(x ** 32)\n'
>>> f.readline()                 # Returns empty string at end-of-file
''
```

However, files also have a method named `__next__` in 3.X (and `next` in 2.X) 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('script2.py')       # __next__ loads one line on each call too
>>> f.__next__()                 # But raises an exception at end-of-file
'import sys\n'
>>> f.__next__()                 # Use f.next() in 2.X, or next(f) in 2.X or 3.X
'print(sys.path)\n'
>>> f.__next__()
'x = 2\n'
>>> f.__next__()
'print(x ** 32)\n'
>>> f.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

This interface is most of 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 an iterator 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 `StopIt eration` exception to determine when to exit. As we'll see in a moment, for some objects the full protocol includes an additional first step to call `iter`, but this isn't required for files.

The net effect of this magic is that, as mentioned in Chapter 9 and Chapter 13, 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('script2.py'):          # Use file iterators to read by lines
...     print(line.upper(), end='')          # Calls __next__, catches StopIteration
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(X ** 32)
```

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; in 2.X, a trailing comma works the same as the `end`). 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('script2.py').readlines():
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(X ** 32)
```

This `readlines` technique still works but 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

As mentioned in the prior chapter's sidebar, "Why You Will Care: File Scanners" on page 400, it's also possible to read a file line by line with a `while` loop:

```
>>> f = open('script2.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. Anytime we trade Python code for C code, speed tends to increase. This is not an absolute truth, though, especially in Python 3.X; we'll see timing techniques later in Chapter 21 for measuring the relative speed of alternatives like these.[1]

Version skew note: In Python 2.X, the iteration method is named `X.next()` instead of `X.__next__()`. For portability, a `next(X)` built-in function is also available in both Python 3.X and 2.X (2.6 and later), and calls `X.__next__()` in 3.X and `X.next()` in 2.X. Apart from method names, iteration works the same in 2.X and 3.X in all other ways. In 2.6 and 2.7, simply use `X.next()` or `next(X)` for manual iterations instead of 3.X's `X.__next__()`; prior to 2.6, use `X.next()` calls instead of `next(X)`.

Manual Iteration: iter and next

To simplify manual iteration code, Python 3.X also provides a built-in function, `next`, that automatically calls an object's `__next__` method. Per the preceding note, this call also is supported on Python 2.X for portability. Given an iterator object `X`, the call `next(X)` is the same as `X.__next__()` on 3.X (and `X.next()` on 2.X), but is noticeably simpler and more version-neutral. With files, for instance, either form may be used:

```
>>> f = open('script2.py')
>>> f.__next__()                        # Call iteration method directly
'import sys\n'
>>> f.__next__()
'print(sys.path)\n'

>>> f = open('script2.py')
>>> next(f)                             # The next(f) built-in calls f.__next__() in 3.X
'import sys\n'
>>> next(f)                             # next(f) => [3.X: f.__next__()], [2.X: f.next()]
'print(sys.path)\n'
```

Technically, there is one more piece to the iteration protocol alluded to earlier. When the `for` loop begins, it first obtains an iterator from the iterable object by passing it to the `iter` built-in function; the object returned by `iter` in turn has the required `next` method. The `iter` function internally runs the `__iter__` method, much like `next` and `__next__`.

1. Spoiler alert: the file iterator still appears to be slightly faster than `readlines` and at least 30% faster than the `while` loop in both 2.7 and 3.3 on tests I've run with this chapter's code on a 1,000-line file (`while` is twice as slow on 2.7). The usual benchmarking caveats apply—this is true only for my Pythons, my computer, and my test file, and Python 3.X complicates such analyses by rewriting I/O libraries to support Unicode text and be less system-dependent. Chapter 21 covers tools and techniques you can use to time these loop statements on your own.

The full iteration protocol

As a more formal definition, Figure 14-1 sketches this full iteration protocol, used by every iteration tool in Python, and supported by a wide variety of object types. It's really based on *two objects*, used in two distinct steps by iteration tools:

- The *iterable* object you request iteration for, whose `__iter__` is run by `iter`
- The *iterator* object returned by the iterable that actually produces values during the iteration, whose `__next__` is run by `next` and raises `StopIteration` when finished producing results

These steps are orchestrated automatically by iteration tools in most cases, but it helps to understand these two objects' roles. For example, in some cases these two objects are the *same* when only a single scan is supported (e.g., files), and the *iterator* object is often temporary, used internally by the iteration tool.

Moreover, some objects are *both* an iteration context tool (they iterate) and an iterable object (their results are iterable)—including Chapter 20's generator expressions, and `map` and `zip` in Python 3.X. As we'll see ahead, more tools become iterables in 3.X—including `map`, `zip`, `range`, and some dictionary methods—to avoid constructing result lists in memory all at once.

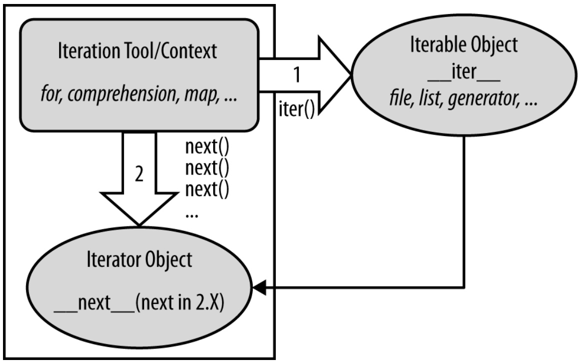

Figure 14-1. The Python iteration protocol, used by for loops, comprehensions, maps, and more, and supported by files, lists, dictionaries, Chapter 20's generators, and more. Some objects are both iteration context and iterable object, such as generator expressions and 3.X's flavors of some tools (such as map and zip). Some objects are both iterable and iterator, returning themselves for the iter() call, which is then a no-op.

In actual code, the protocol's first step 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 from an iterable
>>> I.__next__()                   # Call iterator's next to advance to next item
1
```

```
>>> I.__next__()                    # Or use I.next() in 2.X, next(I) in either line
2
>>> I.__next__()
3
>>> I.__next__()
...error text omitted...
StopIteration
```

This initial step is not required for files, because a file object is its own iterator. Because they support just one iteration (they can't seek backward to support multiple active scans), files have their own `__next__` method and do not need to return a different object that does:

```
>>> f = open('script2.py')
>>> iter(f) is f
True
>>> iter(f) is f.__iter__()
True
>>> f.__next__()
'import sys\n'
```

Lists and many other built-in objects, though, are not their own iterators because they do support multiple open iterations—for example, there may be multiple iterations in nested loops all at different positions. 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
```

Manual iteration

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:[2]

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

2. Technically speaking, the `for` loop calls the internal equivalent of `I.__next__`, instead of the `next(I)` used here, though there is rarely any difference between the two. Your manual iterations can generally use either call scheme.

```
>>> I = iter(L)                    # Manual iteration: what for loops usually do
>>> while True:
...     try:                       # try statement catches exceptions
...         X = next(I)            # Or call I.__next__ in 3.X
...     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 met exceptions briefly in Chapter 11 but will explore them 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, but prefer the iteration protocol if it's used. We'll defer that story until we study class operator overloading in Chapter 30.

Other Built-in Type Iterables

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
b 2
c 3
```

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

```
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'b'
>>> next(I)
'c'
>>> next(I)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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
b 2
c 3
```

We can't delve into their details here, but other Python object types also support the iteration 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, which we met in the preceding chapter) are iterable as well:

```
>>> import os
>>> P = os.popen('dir')
>>> P.__next__()
' Volume in drive C has no label.\n'
>>> P.__next__()
' Volume Serial Number is D093-D1F7\n'
>>> next(P)
TypeError: _wrap_close object is not an iterator
```

Notice that `popen` objects themselves support a `P.next()` method in Python 2.X. In 3.X, they support the `P.__next__()` method, but not the `next(P)` built-in. Since the latter is defined to call the former, this may seem unusual, though both calls work correctly if we use the full iteration protocol employed automatically by `for` loops and other iteration contexts, with its top-level `iter` call (this performs internal steps required to also support `next` calls for this object):

```
>>> P = os.popen('dir')
>>> I = iter(P)
>>> next(I)
' Volume in drive C has no label.\n'
>>> I.__next__()
' Volume Serial Number is D093-D1F7\n'
```

Also in the systems domain, the standard directory walker in Python, `os.walk`, is similarly iterable, but we'll save an example until Chapter 20's coverage of this tool's basis —generators and `yield`.

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.X
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]
```

Note that the `list` call here is not required in 2.X (where `range` builds a real list), and is not needed in 3.X for contexts where iteration happens automatically (such as within `for` loops). It is needed for displaying values here in 3.X, though, and may also be required when list-like behavior or multiple scans are required for objects that produce results on demand in 2.X or 3.X (more on this ahead).

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 0x00000000029B7678>
>>> 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 Detailed Look

Now that we've seen how the iteration protocol works, let's turn to one of its most common use cases. 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 coding patterns 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 similar, 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
[31, 32, 33, 34, 35]
```

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 useful in many contexts. Moreover, depending on your Python and code, list comprehensions might 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 often a major performance advantage to using this expression.

Using List Comprehensions on Files

Let's work through another common application of 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('script2.py')
>>> lines = f.readlines()
>>> lines
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\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?

Anytime 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 \n terminated, and this may not always be the case for the last line in a file):

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(x ** 32)']
```

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 handler 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('script2.py')]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(x ** 32)']
```

This expression does a lot implicitly, but we're getting a lot of work for free here—Python scans the file by lines 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 may be faster than an equivalent for statement, and won't load a file into memory all at once like some other techniques. Again, especially for large files, the 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. To illustrate, here's the list comprehension equivalent to the file iterator uppercase example we met earlier, along with a few other representative operations:

```
>>> [line.upper() for line in open('script2.py')]
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']

>>> [line.rstrip().upper() for line in open('script2.py')]
['IMPORT SYS', 'PRINT(SYS.PATH)', 'X = 2', 'PRINT(X ** 32)']

>>> [line.split() for line in open('script2.py')]
[['import', 'sys'], ['print(sys.path)'], ['x', '=', '2'], ['print(x', '**', '32)']]

>>> [line.replace(' ', '!') for line in open('script2.py')]
['import!sys\n', 'print(sys.path)\n', 'x!=!2\n', 'print(x!**!32)\n']

>>> [('sys' in line, line[:5]) for line in open('script2.py')]
[(True, 'impor'), (True, 'print'), (False, 'x = 2'), (False, 'print')]
```

Recall that 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. The last of these shows how we can also collect *multiple* results, as long as they're wrapped in a collection like a tuple or list.

One fine point here: recall from Chapter 9 that file objects *close* themselves automatically when garbage-collected if still open. Hence, these list comprehensions will also automatically close the file when their temporary file object is garbage-collected after the expression runs. Outside CPython, though, you may want to code these to close manually if this is run in a loop, to ensure that file resources are freed immediately. See Chapter 9 for more on file close calls if you need a refresher on this.

Extended List Comprehension Syntax

In fact, list comprehensions can be even richer in practice, and even constitute a sort of *iteration mini-language* in their fullest forms. Let's take a quick look at their syntax tools here.

Filter clauses: if

As one particularly useful extension, the `for` loop nested in a comprehension 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('script2.py') if line[0] == 'p']
>>> lines
['print(sys.path)', 'print(x ** 32)']
```

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('script2.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
...
>>> res
['print(sys.path)', 'print(x ** 32)']
```

This `for` statement equivalent works, but it takes up four lines instead of one and may run slower. In fact, you can squeeze a substantial amount of logic into a list comprehension when you need to—the following works like the prior but selects only lines that *end in a digit* (before the newline at the end), by filtering with a more sophisticated expression on the right side:

```
>>> [line.rstrip() for line in open('script2.py') if line.rstrip()[-1].isdigit()]
['x = 2']
```

As another `if` filter example, the first result in the following gives the total lines in a text file, and the second strips whitespace on both ends to *omit blank links* in the tally in just one line of code (this file, not included, contains lines describing typos found in the first draft of this book by my proofreader):

```
>>> fname = r'd:\books\5e\lp5e\draft1typos.txt'
>>> len(open(fname).readlines())                               # All lines
263
>>> len([line for line in open(fname) if line.strip() != ''])  # Nonblank lines
185
```

Nested loops: for

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.

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 all the *ordered combinations* 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.

Because comprehensions are generally best taken in multiple doses, we'll cut this story short here for now. We'll revisit list comprehensions in Chapter 20 in the context of functional programming tools, and will define their syntax more formally and explore additional examples there. As we'll find, comprehensions turn out to be just as related to *functions* as they are to looping *statements*.

A blanket qualification for all *performance claims* in this book, list comprehension or other: the relative speed of code depends much on the exact code tested and Python used, and is prone to change from release to release.

For example, in CPython 2.7 and 3.3 today, list comprehensions can still be twice as fast as corresponding `for` loops on some tests, but just marginally quicker on others, and perhaps even slightly slower on some when `if` filter clauses are used.

We'll see how to time code in Chapter 21, and will learn how to interpret the file *listcomp-speed.txt* in the book examples package, which times this chapter's code. For now, keep in mind that absolutes in performance benchmarks are as elusive as consensus in open source projects!

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* built-in 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('script2.py'):         # Use file iterators
...     print(line.upper(), end='')
```

```
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(X ** 32)
```

But also much more. For instance, list comprehensions and the `map` built-in function use the same protocol as their `for` loop cousin. When applied to a file, they both leverage the file object's iterator automatically to scan line by line, fetching an iterator with `__iter__` and calling `__next__` each time through:

```
>>> uppers = [line.upper() for line in open('script2.py')]
>>> uppers
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']

>>> map(str.upper, open('script2.py'))        # map is itself an iterable in 3.X
<map object at 0x00000000029476D8>
>>> list(map(str.upper, open('script2.py')))
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']
```

We introduced the `map` call used here briefly in the preceding chapter (and in passing in Chapter 4); 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.X, 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 Chapter 19 and Chapter 20.

Many of Python's other built-ins process iterables, too. For example, `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.X, like `map`. Here they are in action running the file's iterator automatically to read line by line:

```
>>> sorted(open('script2.py'))
['import sys\n', 'print(sys.path)\n', 'print(x ** 32)\n', 'x = 2\n']

>>> list(zip(open('script2.py'), open('script2.py')))
[('import sys\n', 'import sys\n'), ('print(sys.path)\n', 'print(sys.path)\n'),
('x = 2\n', 'x = 2\n'), ('print(x ** 32)\n', 'print(x ** 32)\n')]

>>> list(enumerate(open('script2.py')))
[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'),
(3, 'print(x ** 32)\n')]

>>> list(filter(bool, open('script2.py')))     # nonempty=True
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

>>> import functools, operator
>>> functools.reduce(operator.add, open('script2.py'))
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\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 their details for now; the point to notice here is their use of the iteration protocol for files and other iterables.

We first saw the `sorted` function used here at work in, 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.X instead of an iterable.

Interestingly, the iteration protocol is even more pervasive in Python today than the examples so far have demonstrated—essentially *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 tools such as the `list` and `tuple` built-in functions (which build new objects from iterables), and the string `join` method (which makes a new string by putting a substring between strings contained in an iterable). Consequently, these will also work on an open file and automatically read one line at a time:

```
>>> list(open('script2.py'))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

>>> tuple(open('script2.py'))
('import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n')

>>> '&&'.join(open('script2.py'))
'import sys\n&&print(sys.path)\n&&x = 2\n&&print(x ** 32)\n'
```

Even some tools you might not expect fall into this category. For example, sequence assignment, the `in` membership test, slice assignment, and the list's `extend` method also leverage the iteration protocol to scan, and thus read a file by lines automatically:

```
>>> a, b, c, d = open('script2.py')         # Sequence assignment
>>> a, d
('import sys\n', 'print(x ** 32)\n')

>>> a, *b = open('script2.py')              # 3.X extended form
>>> a, b
('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n'])

>>> 'y = 2\n' in open('script2.py')         # Membership test
False
>>> 'x = 2\n' in open('script2.py')
True

>>> L = [11, 22, 33, 44]                    # Slice assignment
>>> L[1:3] = open('script2.py')
>>> L
[11, 'import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n', 44]

>>> L = [11]
>>> L.extend(open('script2.py'))            # list.extend method
>>> L
[11, 'import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']
```

Per Chapter 8 `extend` iterates automatically, but `append` does not—use the latter (or similar) to add an iterable to a list without iterating, with the potential to be iterated across later:

```
>>> L = [11]
>>> L.append(open('script2.py'))              # list.append does not iterate
>>> L
[11, <_io.TextIOWrapper name='script2.py' mode='r' encoding='cp1252'>]
>>> list(L[1])
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']
```

Iteration is a broadly supported and powerful model. Earlier, we saw that the built-in `dict` call accepts an iterable `zip` result, too (see Chapter 8 and Chapter 13). For that matter, so does the `set` call, as well as the newer set and dictionary comprehension expressions in Python 3.X and 2.7, which we met in Chapter 4, Chapter 5, and Chapter 8:

```
>>> set(open('script2.py'))
{'print(x ** 32)\n', 'import sys\n', 'print(sys.path)\n', 'x = 2\n'}

>>> {line for line in open('script2.py')}
{'print(x ** 32)\n', 'import sys\n', 'print(sys.path)\n', 'x = 2\n'}

>>> {ix: line for ix, line in enumerate(open('script2.py'))}
{0: 'import sys\n', 1: 'print(sys.path)\n', 2: 'x = 2\n', 3: 'print(x ** 32)\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('script2.py') if line[0] == 'p'}
{'print(x ** 32)\n', 'print(sys.path)\n'}
>>> {ix: line for (ix, line) in enumerate(open('script2.py')) if line[0] == 'p'}
{1: 'print(sys.path)\n', 3: 'print(x ** 32)\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. Later in the book we'll meet a relative of comprehensions—generator expressions—that deploys the same syntax and works on iterables too, but is also iterable itself:

```
>>> list(line.upper() for line in open('script2.py'))          # See Chapter 20
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']
```

Other built-in functions support the iteration protocol as well, but frankly, some 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('script2.py'))                   # Line with max/min string value
'x = 2\n'
>>> min(open('script2.py'))
'import sys\n'
```

There's one last iteration context that's worth mentioning, although it's mostly 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 this call syntax; Chapter 20 for a section that extends this idea to generator expressions; and Chapter 11 for tips on using the following's 3.X `print` in 2.X as usual):

```
>>> 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('script2.py'))              # Iterates by lines too!
import sys
&print(sys.path)
&x = 2
&print(x ** 32)
```

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.X as well, we need to move on to the next section.

New Iterables in Python 3.X

One of the fundamental distinctions of Python 3.X is its stronger emphasis on iterators than 2.X. This, along with its Unicode model and mandated new-style classes, is one of 3.X's most sweeping changes.

Specifically, 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.X, 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 iterables and process them. All of these tools produce results on demand in Python 3.X, instead of constructing result lists as they do in 2.X.

Impacts on 2.X Code: Pros and Cons

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 some function and method call results in a `list(...)` call in order to force them to produce all their results at once for *display*:

```
>>> zip('abc', 'xyz')                   # An iterable in Python 3.X (a list in 2.X)
<zip object at 0x000000000294C308>

>>> list(zip('abc', 'xyz'))             # Force list of results in 3.X to display
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

A similar conversion is required if we wish to apply list or *sequence operations* to most iterables that generate items on demand—to index, slice, or concatenate the iterable itself, for example. The list results for these tools in 2.X support such operations directly:

```
>>> Z = zip((1, 2), (3, 4))             # Unlike 2.X lists, cannot index, etc.
>>> Z[0]
TypeError: 'zip' object is not subscriptable
```

As we'll see in more detail in Chapter 20, conversion to lists may also be more subtly required to support *multiple iterations* for newly iterable tools that support just one scan such as map and zip—unlike their 2.X list forms, their values in 3.X are exhausted after a single pass:

```
>>> M = map(lambda x: 2 ** x, range(3))
>>> for i in M: print(i)
...
1
2
4
>>> for i in M: print(i)              # Unlike 2.X lists, one pass only (zip too)
...
>>>
```

Such conversion isn't required in 2.X, because functions like zip return lists of results. In 3.X, though, they return iterable objects, producing results on demand. This may break 2.X code, and 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.X iterables in action.

The range Iterable

We studied the range built-in's basic behavior in the preceding chapter. In 3.X, it returns an iterable 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:\code> c:\python33\python
>>> R = range(10)               # range returns an iterable, not a list
>>> R
range(0, 10)

>>> I = iter(R)                 # Make an iterator from the range iterable
>>> 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.X 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: As first mentioned in the preceding chapter, 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.X, `xrange` is no longer available in 3.X—it has been subsumed. You may still both see and use 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 the prior chapter, the `file.xreadlines()` method used to minimize memory use in 2.X has been dropped in Python 3.X for similar reasons, in favor of file iterators.

The map, zip, and filter Iterables

Like `range`, the `map`, `zip`, and `filter` built-ins also become iterables in 3.X 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.X. 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 iterables, 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 iterable, not a list
>>> M
<map object at 0x00000000029B75C0>
>>> 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 iterable/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, is an iteration context itself, but also returns an iterable with an iterator that works the same way:

```
>>> Z = zip((1, 2, 3), (10, 20, 30))    # zip is the same: a one-pass iterator
>>> Z
<zip object at 0x0000000002951108>

>>> 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))    # Manual iteration (iter() not needed)
>>> next(Z)
(1, 10)
>>> next(Z)
(2, 20)
```

The `filter` built-in, which we met briefly in Chapter 12 and will 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, and `bool` returns an object's truth value):

```
>>> filter(bool, ['spam', '', 'ni'])
<filter object at 0x00000000029B7B70>
>>> 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.X. It can also generally be emulated by extended list comprehension syntax that automatically tests truth values:

```
>>> [x for x in ['spam', '', 'ni'] if bool(x)]
['spam', 'ni']
>>> [x for x in ['spam', '', 'ni'] if x]
['spam', 'ni']
```

Multiple Versus Single Pass Iterators

It's important 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, in 3.X `zip`, `map`, and `filter` do not support multiple active iterators on the same result; because of this the `iter` call is optional for stepping through such objects' results—their `iter` is themselves (in 2.X these built-ins return multiple-scan lists so the following does not apply):

```
>>> 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)                               # (3.X) 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)                               # (3.X) Single scan is exhausted!
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)                               # Multiple active scans, like 2.X lists
0
```

When we code our own iterable objects with classes later in the book (Chapter 30), 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 just a single active iteration scan. In that chapter, we'll see some subtle implications of one-shot iterators in loops that attempt to scan multiple times—code that formerly treated these as lists may fail without manual list conversions.

Dictionary View Iterables

Finally, as we saw briefly in Chapter 8, in Python 3.X 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. Views are also available in 2.7 as an option, but under special method names to avoid impacting existing code. 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 iterables here's the rest of this story—in Python 3.3 (your key order may vary):

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'b': 2, 'c': 3}

>>> K = D.keys()                          # A view object in 3.X, not a list
>>> K
dict_keys(['a', 'b', 'c'])

>>> next(K)                               # Views are not iterators themselves
TypeError: dict_keys object is not an iterator

>>> I = iter(K)                           # View iterables have an iterator,
>>> next(I)                               # which can be used manually,
'a'                                       # but does not support len(), index
>>> next(I)
'b'

>>> for k in D.keys(): print(k, end=' ')  # All iteration contexts use auto
...
a b c
```

As for all iterables that produce values on request, you can always force a 3.X 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', 'b', 'c']

>>> V = D.values()                   # Ditto for values() and items() views
>>> V
dict_values([1, 2, 3])
>>> list(V)                          # Need list() to display or index as list
[1, 2, 3]

>>> V[0]
```

```
TypeError: 'dict_values' object does not support indexing
>>> list(V)[0]
1

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

>>> for (k, v) in D.items(): print(k, v, end=' ')
...
a 1 b 2 c 3
```

In addition, 3.X dictionaries still are iterables themselves, with an iterator that returns successive keys. Thus, it's not often necessary to call `keys` directly in this context:

```
>>> D                                          # Dictionaries still produce an iterator
{'a': 1, 'b': 2, 'c': 3}                       # Returns next key on each iteration
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'b'

>>> for key in D: print(key, end=' ')          # Still no need to call keys() to iterate
...                                            # But keys is an iterable in 3.X too!
a b c
```

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.X. 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. We saw this in Chapter 8, but it's important enough to 2.X programmers making the switch to demonstrate again:

```
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> for k in sorted(D.keys()): print(k, D[k], end=' ')
...
a 1 b 2 c 3
>>> for k in sorted(D): print(k, D[k], end=' ')      # "Best practice" key sorting
...
a 1 b 2 c 3
```

Other Iteration Topics

As mentioned in this chapter's introduction, there is more coverage of both list comprehensions and iterables in Chapter 20, in conjunction with functions, and again in Chapter 30 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 iterables defined with classes allow arbitrary objects and operations to be used in any of the iteration contexts we've met in this chapter. By supporting just a single operation—*iteration*—objects may be used in a wide variety of contexts and tools.

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.X.

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. Though a bit of a diversion from the more detailed aspects of coding, 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 iterable objects 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 iterable object across which it is iterating. It first fetches an iterator from the iterable by passing the object to `iter`, and then calls this iterator object's `__next__` method in 3.X on each iteration and catches the `StopIteration` exception to determine when to stop looping. The method is named `next` in 2.X, and is run by the `next` built-in function in both 3.x and 2.X. Any object that supports this model works in a `for` loop and

in all other iteration contexts. For some objects that are their own iterator, the initial `iter` call is extraneous but harmless.

2. Both are iteration tools and contexts. List comprehensions are a concise and often 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 (see answer #1) 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 tool 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` handler method on each iteration. This approach is generally best in terms of coding simplicity, memory space, and possibly execution speed requirements.
5. I'll accept any of the following as correct answers: fear, intimidation, nice red uniforms, a comfy chair, and soft pillows.